Efficient Model Checking: The Power of Randomness

Kiviriga, Andrej

DOI (link to publication from Publisher):
10.54337/aau534292082

Publication date:
2023

Document Version
Publisher's PDF, also known as Version of record

EFFICIENT MODEL CHECKING: THE POWER OF RANDOMNESS

BY
ANDREJ KIVIRIGA

DISSERTATION SUBMITTED 2023

AALBORG UNIVERSITY
DENMARK

Efficient Model Checking: The Power of Randomness

Ph.D. Dissertation

Andrej Kiviriga

Dissertation submitted February, 2023

Dissertation submitted: February, 2023

PhD supervisor: Professor Kim Guldstrand Larsen
Aalborg University

PhD Co-supervisor: Associate Professor Ulrik Nyman
Aalborg University

PhD committee: Associate Professor Martin Zimmermann (chair)
Aalborg University, Denmark

Associate Professor Cristina Seceleanu
Mälardalen University, Sweden

Professor Dr. Habil Martin Leucker
University of Lübeck, Germany

PhD Series: Technical Faculty of IT and Design, Aalborg University

Department: Department of Computer Science

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-740-6

Published by:
Aalborg University Press
Kroghstræde 3
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Stibo Complete, 2023

Dedicated to the memory of my beloved grandfather.
I am eternally grateful you have persuaded me to undertake this adventure
and heartbroken I cannot share it with you anymore.

Присвячується пам'яті мого улюбленого дідуся.
Я нескінченно вдячний, що ти переконав мене піти на цю пригоду, і дуже
шкодую, що більше не можу поділитися цим з тобою.

Посвящается памяти моего любимого дедушки.
Я бесконечно благодарен, что ты убедил меня пойти на это приключение,
и очень сожалею, что больше не могу поделиться им с тобой.

Abstract

In modern, computerized world, many safety and business critical operations are now performed by embedded software and the need for correct, safe and optimal systems continues to grow. Unlike testing procedures, the method of model checking allows us to provide guarantees on correctness and optimality of such systems. The main obstacle that impedes the widespread success of model checking is that of the state space explosion – a problem that quickly exposes the limits of verification algorithms when dealing with large and industrial-sized systems.

Apart from proving the correctness of systems, model checking can also be used for falsification of requirements and generating error traces for debugging. In this thesis, we attempt to improve the applicability of falsification and optimization model checking techniques for three problem domains in large safety critical systems – correctness, refinement and optimization. More specifically, we exploit *randomness* and use it at various steps of our methods to reduce computational demands. As the result, we provide lightweight, efficient and scalable falsification and optimization techniques that in many cases significantly reduce time and memory requirements, and in some cases expand the type of models which can be analyzed.

The body of this thesis consists of two parts. In the first part we give an overview of model checking and motivate the necessity for new techniques. We also give a brief introduction of the modelling formalisms relevant to our scope: different extensions of Timed Automata. The second, main part of the thesis consists of four papers. The first paper presents a randomized refinement checking algorithm for an efficient falsification of the refinement relation between two specifications expressed as Timed I/O Automata. In the second paper, a randomized reachability analysis for Timed and Stopwatch Automata is developed, implemented and made available in the model checker Uppaal. A large number of models are used to showcase the strengths of our method, including the model of the industrial-sized Herschel-Planck satellite system. In the third paper, we adapt Monte Carlo Tree Search for verification of optimization problems for models expressed as Priced Timed Automata. The final paper presents a framework for usage-

aware falsification of Cyber-physical systems. The framework computes a list of counterexamples that are ranked according to their probability.

Resumé

I en moderne, computeriseret verden udføres mange sikkerheds- og forretningskritiske operationer nu af indlejret software, og behovet for korrekte, sikre og optimale systemer fortsætter med at vokse. Model checking giver i modsætning til testprocedurer mulighed for at give garantier for korrektheden og optimaliteten af sådanne systemer. Den største hindring for at model cheking bliver en udbredt success, er tilstandsrumeksplosion (state space explosion) - et problem, der hurtigt afslører grænserne for model checking, når man har med store og industrielle systemer at gøre.

Udover at bevise systemers korrekthed, kan model checking også bruges til falsificering af krav og generering af fejlspor til debugging. I denne afhandling forsøges det at øge anvendeligheden af falsificering og optimerings model checking metoder på tre problemdomæner i store sikkerhedskritiske systemer - korrekthed, forfining og optimering. Tilfældighed bliver forsøgt anvendt i forskellige trin af metoderne til at reducere mængden af nødvendige beregning. Dette resulterer i letvægts, effektive og skalerbare falsificerings og optimerings teknikker, der i mange tilfælde reducerer kravene til verifikationstid og hukommelse markant og udvider mængden af modeller, der kan analyseres.

Hoveddelen af denne afhandling består af to dele. Første del giver en oversigt over model checking og motiverer nødvendigheden af nye teknikker. De modelleringssformalismer der er relevante for afhandlingen introduceres også. Disse er alle forskellige udvidelser af tidsautomater (Timed Automata). Den anden hoveddel af afhandlingen består af fire artikler. Den første artikel præsenterer en randomiseret raffineringskontrolalgoritme (refinement checking algorithm) til effektiv falsificering af raffineringsrelationen (the refinment relation) mellem to specifikationer udtrykt som I/O tidsautomater (Timed I/O Automata). I den anden artikel er en randomiseret reachability analyse for tidsautomater (Timed Automata) og stopursautomater (Stopwatch Automata) blevet udviklet, implementeret og gjort tilgængelig i modelcheckeren Uppaal. Et stort antal modeller bruges til at vise metodens styrker, herunder modellen af det industrielle Herschel-Planck satellitsystem. I den tredje artikel tilpasses Monte Carlo Tree Search til verifikation af optimer-

ingsproblemer for modeller udtrykt som tidsautometer med omkostninger (Priced Timed Automata). Den sidste artikel præsenterer en metode, der anvender brugsmøstre til fejlfinding i indlejrede systemer (Cyber-physical systems). Metoden beregner en liste af fejlscenarier, der er rangeret efter deres sandsynlighed.

Contents

Acknowledgements

I would like to express my gratitude to my supervisor Kim G. Larsen and co-supervisor Ulrik Nyman for all the given opportunities and guidance throughout my journey as a PhD student. This thesis would not have been possible, had it not been for your support and invaluable lessons. Thank you for being there when I needed you the most.

A big thanks to Dejan Nickovic for great hospitality and supervision during my external stay in Vienna. I truly enjoyed our collaboration, which I hope we will continue.

For the invaluable help with the implementations inside Uppaal as well as many other crucial lessons that only you could give, I attribute part of my achievement to Marius Mikučionis and Peter Gjøl Jensen.

It is hard to underestimate many fascinating discussions and countless laughs we had over the years with you, Martin Kristjansen. I am grateful to have had you as my office-buddy and I am sure my time as a PhD student would have been boring without you.

Andrej Kiviriga
Aalborg University, February 24, 2023

Part I

Introduction

Introduction

Over the last decades computers have been integrated into our society and it is hard to imagine the world without them. By now countless systems exist to satisfy various needs. However, there is a number of challenges that come along with such technological progress.

Firstly, systems tend to contain errors that can make them unsafe and unreliable. Despite all the effort of software engineering practices, increasing complexity of systems also entails increasing difficulty of error discovery and prevention. Therefore, systems' *correctness* is crucial; it is vital to insure the strict accordance of the system to the intended behavior, especially for *critical systems* found in medical equipment, nuclear power plants, aircraft and satellites.

Secondly, system compatibility is an important challenge. With multiple existing systems, it has become a successful strategy to assemble smaller components into multiplex systems and tailor them according to the user needs. In practice, these components tend to be developed independently and might not be suitable for being combined into larger systems. More concretely, in the reasoning about system compatibility, we are interested in the *refinement* feature that allows one to compare components and safely replace one component by another one in a large system design.

And thirdly, the optimality of a system is an increasing concern in the light of such global problems as the climate change and economic crises. A system might contain no errors, but make an inefficient use of the resources, such as time and money, as well as result in unnecessary high consumption of electricity or emission of gases, to mention a few examples. Such class of problems is known as *optimization* problems. These three problem classes – correctness, refinement and optimization – are gathering significant attention and can be addressed in several ways.

For the first two challenges – correctness and refinement – the most widely known approach is that of *testing*. It allows one to focus resources on "important" parts of the system and is usually used as a *falsification* approach. Carefully crafted testing procedure can help to eliminate most, if not all, issues in the system and for many applications that is enough. Unfortunately,

the major drawback of the testing is its lack of guarantees on correctness of the system as testing procedures are designed and carried out by humans who naturally tend to make mistakes. In safety-critical systems such an error-prone approach cannot be relied on.

To address this, the approach of *formal verification* can be used. It allows reasoning about the correctness of a system with respect to some formal requirements. In particular, interesting to us is the method of *model checking* that was pioneered in the 1980s [38, 39, 55, 111]. Model checking is a method of checking a finite-state model of the system against properties typically expressed in logical formula. In these terms, the model can either be exact, representing an existing implementation of the system, or abstract, defining an overall specification. Model checking lends itself to automation and uses mathematical algorithms to perform exploration of the model. This allows one to provide guarantees, while reasoning about satisfaction of the requirements, for any of the three mentioned problem classes: correctness, refinement and optimization. In recent years formal methods are becoming increasingly popular and are by now used in such tech "giants" as Amazon [23, 103] and Meta [105]. In these companies, formal methods help to prevent subtle but serious bugs from reaching production - bugs which would not be found by any other technique.

Ever since model checking has emerged, a number of different modelling and logic formalisms have been developed that can capture different types of problems and respective requirements. Each formalism has its own trade-off in terms of complexity, captured features and applicable methods. For all of them, a problem known as *state space explosion* is the most prominent obstacle that refers to an exponential growth of the state space as the size of the model increases. As a consequence, the wide use of automated verification in the industry is hampered. To combat this problem and reduce the computational demands, an immense number of techniques have been put through in the last three decades, giving rise to such model checking tools as Uppaal [17], SPIN [76], Kronos [27], Prism [91], NuSMV [37] and more. However, despite all the effort, model checking still remains an expensive approach.

An important observation to be made in the setting of verification is that not all states in the state space are equally important. While some types of requirements demand the entire state space to be searched to provide an answer, other requirements can be concluded on early, once some *target* behavior is discovered. For example, reaching a state that violates a safety requirement allows us to conclude the verification of a safety property without the need to proceed with the rest of the state space. However, in many industrial systems the target behavior constitutes a very small fraction of the state space and is difficult to find. With ever growing systems, the state space explosion is still an ongoing problem that can quickly expose limits of model checking – a problem that will probably never go away.

A very promising way to tackle the state space explosion is by using *randomness*. Deterministic or probabilistic choices in model checking algorithms can instead be replaced with random or semi-random decisions. Randomness can help to discover the target behavior of the system quickly and efficiently, allowing to reason about the imposed requirements well before the whole state space is unfolded. The reasons for that are a few. Random choices are typically much cheaper than computationally expensive decision making techniques, which improves the speed of the search. It also turns out that random variations in the search order can produce huge variations in the result or in the performance of the technique [52]. Last but not least, randomized methods will often consume significantly less memory during the search.

As we shall see, existing research hints that randomness can have a dramatic effect on the performance of algorithms, greatly improving their scalability; therefore, in this thesis we study the potential of employing randomness in different domains of model-checking that include correctness, refinement and optimization problem classes.

Outline

The rest of this chapter is organized as follows. We review the state of the art for model checking, including models, tools, logical formalisms and approaches used to deal with correctness, refinement and optimization problem classes. Next we focus on the explanation of how randomness is employed in some of those approaches. After this follows a summary of contributions from each of the four papers that make up the basis of this thesis.

1 State of the Art

This section gives an overview of the model checking field and relevant to this thesis formal models, logics, tools and approaches.

1.1 Models

In this thesis we primarily focus on timed systems, the central model being that of Timed Automata [8]. It is the type of automaton that supports the notion of time represented by continuous variables that are called *clocks*. All clocks evolve over time at the same rate and can be reset when needed. This allows modelling of a large variety of systems that have a certain timing behavior to them. In addition to Timed Automata, a number of extensions exist that further expand the features that can be captured.

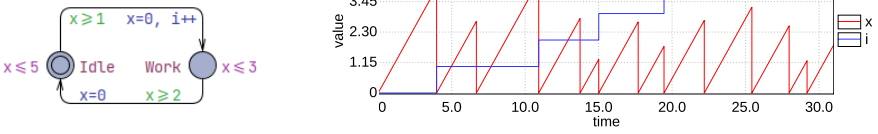
We start by formally defining an Extended Timed Automata – an extension of Timed Automata with integer variables. Let $\mathcal{C}$ be a finite set of clocks and $\mathcal{X}$ be a finite set of integer variables. A *valuation* function is a mapping $v : (\mathcal{C} \cup \mathcal{X}) \rightarrow (\mathbb{R}_{\geq 0} \cup \mathbb{Z})$ such that clocks are mapped to reals $\mathbb{R}_{\geq 0}$ and variables are mapped to integers $\mathbb{Z}$. Let $y \bowtie n$ be a linear constraint where $y \in \mathcal{C} \cup \mathcal{X}$, $n \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$. Let $\mathcal{B}(\mathcal{C}, \mathcal{X})$ be a set of *guards* such that each guard is represented as a finite conjunction of linear constraints. Let $\mathcal{U}$ be a set of all possible update operations in the form $c = 0$ and $x = e$, where $c \in \mathcal{C}$, $x \in \mathcal{X}$ and e is an arithmetic expression over $\mathcal{X}$.

Definition 1
An Extended Timed Automaton (TA) is a tuple $(L, l_0, \mathcal{C}, \mathcal{X}, \Sigma, E, I)$ where:

- L is a finite set of locations,
- l_0 is the initial location,
- $\mathcal{C}$ is a set of clocks,
- $\mathcal{X}$ is a set of variables,
- Σ is a finite set of actions,
- $E \subseteq L \times \mathcal{B}(\mathcal{C}, \mathcal{X}) \times (\Sigma \cup \epsilon) \times 2^{\mathcal{U}} \times L$ is a finite set of edges. An edge is represented as $(l, g, a, u, l') \in E$, connects two locations l and l', and contains a *guard* g, an action a and an *update* u, and
- $I : L \rightarrow \mathcal{B}(\mathcal{C}, \mathcal{X})$ is a set of location invariants.

Example 1.1 (Timed Automaton)
Figure 1 (a) shows an example Timed Automaton with the clock x that

models the behavior of a simple CPU. The two locations `Idle` and `Work` represent two modes where CPU is either idling or processing a task, respectively. Starting in the idle mode, CPU will transition into the working mode after at least 1 and at most 5 time units. The timing constraints are are imposed by the guard `x>=1` on the edge and the invariant `x<=5` on the `Idle` location. From there, CPU is processing a task for a time interval between 2 (`x>=2`) and 3 (`x<=3`) time units and only then can return to the `Idle` location again. Both of the edges have an update `x=0` that resets the clock `x`. In addition, variable `i` acts as an integer counter of the number of times CPU has entered the working state and is incremented (`i++`) every time the edge from `Idle` to `Work` is taken. The dynamics of the clock `x` and the variable `i` can be seen in the simulation of CPU automaton (b).



(a) Timed Automaton

(b) Simulation of Timed Automaton.

Fig. 1: CPU Timed Automaton (a) and its simulation (b).

A Timed Automaton is a syntactic construct whose semantics is given in terms of a Timed Transition System that we now define.

Definition 2
A Timed Transition System (TTS) is a tuple $(S, s_0, \Sigma, \rightarrow)$ where

- S is a set of states,
- s_0 is the initial state,
- Σ is a finite set of actions, and
- $\rightarrow \subseteq S \times (\Sigma \cup \epsilon \cup \mathbb{R}_{\geq 0}) \times S$ is a set of transitions that connect the states. We write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \rightarrow$.

For our TTS, we require both delay and discrete transitions to be deterministic. We also require delay transitions of our TTS to be *time reflexive* such that a zero-delay does not change the state. Lastly, we impose the requirement of *time additivity* so that a larger delay can be split into two smaller

delays that, if taken consecutively, will lead to the same state as the larger delay. The three requirements are formally defined as follows [47]:

- Time determinism: if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ then $s' = s''$,
- Timed reflexivity: $s \xrightarrow{0} s$ for all states of TTS, and
- Time additivity: if $s \xrightarrow{d_1} s' \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$ for all $d_1, d_2 \in \mathbb{R}_{\geq 0}$.

Before we can proceed to definition of Timed Automata semantics, we first need to explain some operations on clock valuations. Given a valuation v we can change the valuations of clocks and variables with an *update* operation $v[u]$, where $u \in 2^{\mathcal{U}}$. Also, given a valuation v and a delay $d \in \mathbb{R}_{\geq 0}$ we can progress in time with *delay* operation such that clocks change their valuations and variables remain the same. More formally, $(v+d)(c) = v(c) + d$ for $c \in \mathcal{C}$ and $(v+d)(x) = v(x)$ for $x \in \mathcal{X}$. With these operations at hand, we can proceed to define the semantics of TA as follows.

Definition 3
The semantics of a TA $A = (L, l_0, \mathcal{C}, \mathcal{X}, \Sigma, E, I)$ is given by a TTS $[\![A]\!]_{sem} = (L \times v(\mathcal{C}, \mathcal{X}), (l_0, \mathbf{0}), \Sigma, \rightarrow)$ where $\mathbf{0}$ is a function that maps all clocks to 0 and all variables to some initial value set by the user such that $\mathbf{0} \models I(l_0)$, Σ is the same action set as in A, and the transition relation $\rightarrow$ is given by the following two rules:

- *delay transition*: $(l, v) \xrightarrow{d} (l, v')$ iff $d \in \mathbb{R}_{\geq 0}$ and $v' = (v + d)$ and $v' \models I(l)$, and
- *discrete transition*: $(l, v) \xrightarrow{a} (l', v')$ iff $\exists (l, g, a, u, l') \in E$, s.t. $v \models g$ and $v' = v[u]$ and $v' \models I(l')$.

A timed run of a Timed Automaton A is a sequence of alternating delay and discrete transitions starting from the initial state s_0:

$$\pi = s_0 \xrightarrow{d_1} s_0' \xrightarrow{a_1} s_1 \ldots \xrightarrow{d_i} s_{i-1}' \xrightarrow{a_i} s_i \ldots$$

Example 1.2 (Timed run)
Below we give an example run for the Timed Automaton from Figure 1 (a):

$$\begin{aligned}\pi_1 = (\texttt{Idle}, x = 0, i = 0) &\xrightarrow{3.5} (\texttt{Idle}, x = 3.5, i = 0) \xrightarrow{\epsilon} (\texttt{Work}, x = 0, i = 1) \\ &\xrightarrow{2} (\texttt{Work}, x = 2, i = 1) \xrightarrow{\epsilon} (\texttt{Idle}, x = 0, i = 1) \xrightarrow{4.22} (\texttt{Idle}, x = 4.22, i = 1) \\ &\xrightarrow{\epsilon} (\texttt{Work}, x = 0, i = 2) \xrightarrow{2.17} (\texttt{Work}, x = 2.17, i = 2) \ldots\end{aligned}$$

The reachability analysis for Timed Automata without discrete variables was shown to be decidable and PSPACE-complete [8]. We note that the addition of unbounded integer variables lifts the expressivity to be Touring-complete and the problem thus becomes undecidable. However, in practice and for tools like UPPAAL, the variables are usually bounded and hence do not influence the decidability of the problem.

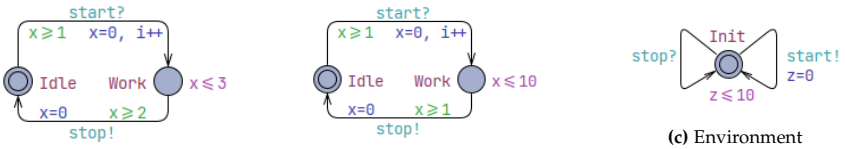
Timed Input/Output Automata

The action set Σ of a Timed Automaton is split into (observable) inputs Σ_i and outputs Σ_o. Syntactically, this is denoted by an postfix ! and ? to the action label for outputs and inputs, respectively. The action set consists of inputs and outputs which are disjoint, i.e. $\Sigma = \Sigma_i \cup \Sigma_o$ and $\Sigma_i \cap \Sigma_o = \emptyset$. There can also be an empty action ϵ that represents an internal (unobservable) action within the automaton. Extending Timed Automata with input and output action separation, we get Timed I/O Automaton [47]. The role of inputs and outputs can be viewed as a way for different components to synchronize their behavior. The input actions are not controlled by the component itself and edges with input actions can only be taken once a corresponding output action is provided by the environment.

We point the reader's attention to the fact that there exists another formalism for Timed I/O Automata and its respective transition system by Lynch et al. [83], in which the used input/ouput model has been initially proposed by Lynch [99]. However, we use the Timed I/O Automata formalism proposed by Larsen et. al [47] instead which shall be viewed as an extension with the addition of *quotient* and *conjunction* operators, and game-based treatment of *refinement*, *composition* and *quotient* operators. There are more differences between the two formalisms of Timed I/O Automata. We refer the interested reader to [47] (and [83]) for all the subtle details on those.

> **Example 1.3 (Timed I/O Automaton)**
> Figure 2 (a) extends our CPU example by replacing previously unobservable actions ϵ with input an output action labels. More specifically, CPU will start processing a task only after receiving an input `start!` (consumed by `start?`), which can happen after at least one time unit is spent idling. Once at `Work`, CPU is in control to decide when the task is finished by outputting on `stop!` (action) channel after between 2 and 3 time units.

An arbitrary number of automata can be parallelly composed into a network of automata. In such network, different components synchronize on inputs and outputs, and all together act as a single system. The input actions of one component can be synchronized with the same label output action of



(a) CPU automaton

(b) CPU automaton

(c) Environment automaton

Fig. 2: Timed I/O Automata examples of CPU (a), (b) and the environment (c).

another component. The synchronization can either be a *handshake*, where only 2 components change their states, or a *broadcast*, where all components that can consume the output advance their states. Here we don't delve into the semantics of composition and different types of synchronization, and instead refer the interested reader to [47, 97] for more details. We, however, need to define the notion of input-enabledness of specifications.

Definition 4
A Timed Transition System is a *specification* if each of its states $s \in S$ is *input-enabled*: $\forall i \in \Sigma_i . \exists s'$ such that $s \xrightarrow{i} s'$.

Input-enabledness supports the belief that the input cannot be prevented from being sent to the system, i.e. the system must be able to consume any input from the input set and do so at any state. If there is no special behavior that a component exhibits on a certain input at a certain time, the input-enabledness is usually ensured by location self-loop transitions with the action label of that input and with necessary guards such that the input is available at all times while respecting the requirement of non-determinism. In practice, such self-loop transitions are typically left out, to improve readability of a syntactic automaton design, and assumed to be implicit.

Example 1.4 (Composition)
The previously examined automaton from Figure 2 (a) can be composed with the component (c) that represents the environment. The environment (c) is then in control of when CPU should start processing a task by broadcasting output `start!`, which can happen only once in 10 time units. If the `start!` action is send by the environment when CPU is not ready to receive it, the action gets consumed by an implicit location self-loop transition.

The specification theory for Timed I/O Automata [47] supports formal comparison between components, allowing to reason about the *refinement* relation between different *specifications* to decide whether one component can be safely replaced by another one in a larger system design.

Definition 5
A specification $K = (S^K, k_o, \Sigma, \rightarrow^K)$ *refines* a specification $T = (S^T, t_o, \Sigma, \rightarrow^T)$, denoted $K \leq T$, iff there exists a binary relation $R \in S^K \times S^T$ containing (k_0, t_0) such that for each pair of states $(k, t) \in R$ the following three conditions hold:

- if $t \xrightarrow{i?}^T t'$ for some $t' \in S^T$ then there must be $k \xrightarrow{i?}^K k'$ for some $k' \in S^K$ and $(k', t') \in R$
- if $k \xrightarrow{o!}^K k'$ for some $k' \in S^K$ then there must be $t \xrightarrow{o!}^T t'$ for some $t' \in S^T$ and $(k', t') \in R$
- if $k \xrightarrow{d}^K k'$ for some $d \in \mathbb{R}_{\geq 0}$ then there must be $t \xrightarrow{d}^T t'$ for some $t' \in \mathbb{R}_{\geq 0}$ and $(k', t') \in R$

Example 1.5 (Refinement)
Consider two Timed I/O Automata (a) and (b) from Figure 2. Both automata have an input action `start?` controlled by some external environment, e.g. such as previously seen component (c), that decides when CPU should start the work. On the other hand, CPU is in charge of an edge with an output action `stop!` which can be executed once the work is done (in accordance with the timing constraints). In fact, automaton (a) refines automaton (b): the invariant of location `Work` and the guard of a `stop!` action edge have been tightened. We now show the refinement relation R between (a) and automaton (b):

$$R = \{\langle(\texttt{Idle}_a, x = v, i = n), (\texttt{Idle}_b, x = v, i = n) \mid v \in \mathbb{R}_{\geq 0} \wedge n \in \mathbb{Z}_{\geq 0}\rangle,$$
$$\langle(\texttt{Work}_a, x = v, i = n), (\texttt{Work}_b, x = v, i = n) \mid 0 \leq v \leq 3 \wedge n \in \mathbb{Z}_{\geq 0}\rangle\}$$

Stopwatch Automata

Allowing to stop and start clocks by setting their evolution rate to either 0 or 1, gives rise to Stopwatch Automata [34]. With the ability to stop and resume clocks it becomes possible to apply a model-based approach to solve such problems as schedulability with preemption. The traditionally used for schedulability analysis Worst Case Response Time approach (WCRT) [33, 81] is known to be over-approximate and might falsely declare system unschedulable. Unlike WCRT, the model-based approach with Stopwatch Automata helps to keep track of more parameters, allowing for less pessimistic and more precise analysis.

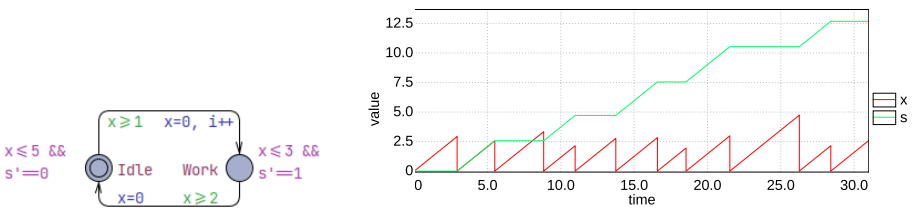
Definition 6
A Stopwatch Automaton (SWA) is a tuple $(L, l_0, \mathcal{C}, \mathcal{X}, \Sigma, E, I, D)$ representing a Timed Automaton extended with D, where $D : L \times \mathcal{C} \to \{0, 1\}$ gives derivatives of clocks at locations, limited to either 0 or 1.

In the underlying semantics of a Stopwatch Automaton, only the *delay transition* is affected:

- $(l, v) \xrightarrow{d} (l, v')$ iff $d \in \mathbb{R}_{\geq 0}$ and $\forall c \in \mathcal{C}.(v'(c) = (v(c) + d \cdot D(l, c))$ and $v' \models I(l)$

Thus, only those clocks progress in time whose derivative is set to 1. In practice, for readability purposes, the derivative of each clock at all locations is assumed to be 1 per default, unless explicitly stated otherwise. Stopwatch Automata was proven [34] to be as expressive as *Linear Hybrid Automata* and shown to be semi-decidable [6].

Example 1.6 (Stopwatch Automaton)
Adding a stopwatch `s` yields a Stopwatch Automaton (a) in Figure 3. The stopwatch measures the time CPU spends working. It progresses normally at location `Work` but is stopped at location `Idle` (`s'==0`). The simulation (b) observes the evolution of the stopwatch occurring only in location `Work`.



(a) Stopwatch Automaton

(b) Simulation of Stopwatch Automaton.

Fig. 3: CPU Stopwatch Automaton (a) and its simulation (b).

Priced Timed Automata

Another extension of Timed Automaton, namely Priced Timed Automata [18], allows us to model resource-consuming systems. In addition to simple clocks, a *cost* variable is present in the model and can evolve according to non-negative integer rates. The rates can depend only on the discrete state of the model, i.e. discrete variables or locations. In essence, the cost variable

is an observing, implicit clock in the automaton that cannot be used in constraints and cannot be reset. Performing certain actions or spending time in certain locations can now have a price. This provides a way to model e.g. different planning problems, as well as search for optimal solutions by either minimizing or maximizing the cost of a witnessing trace.

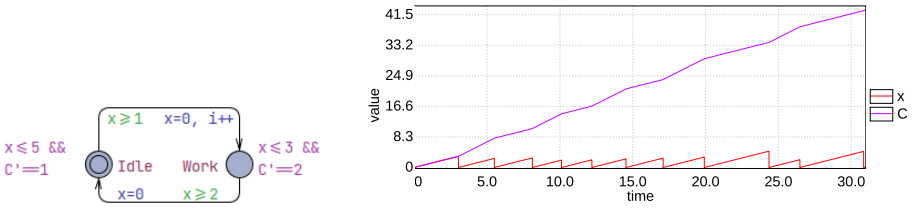
Definition 7
A Priced Timed Automaton (PTA) is a tuple $(L, l_0, \mathcal{C}, \mathcal{X}, \Sigma, E, I, P)$ representing a Timed Automaton extended with P, where $P : (L \cup E) \to \mathbb{Z}_{\geq 0}$ assigns cost rates and cost increments to locations and edges.

In the underlying semantics of a Priced Timed Automaton, both *delay* and *action* transition now have a cost:

- *delay transition*: $(l, v) \xrightarrow{d}_p (l, v')$ iff $d \in \mathbb{R}_{\geq 0}$ and $v' = (v + d)$ and $v' \models I(l)$ and $p = d \cdot P(l)$, and
- *discrete transition*: $(l, v) \xrightarrow{a}_p (l', v')$ iff $\exists (l, g, a, u, l') \in E$, s.t. $v \models g$ and $v' = v[u]$ and $v' \models I(l')$ and $p = P((l, g, a, u, l'))$.

At any point in time, the cost of a timed run in Priced Timed Automaton is the sum of costs of all taken delay and action transitions. In practice, the cost of a Priced Timed Automaton is tracked by an implicit cost variable. The problem of cost-optimal reachability was shown decidable [18], and later proven to be PSPACE complete [25].

Example 1.7 (Priced Timed Automaton)
To track the cost, variable `C` is used that evolves according to non-negative integer rates. This gives a Priced Timed automaton and its simulation in Figure 4 (a) and (b), respectively. Here, the cost of working is twice higher than the cost of idling (represented by cost evolution rates 2 and 1). Both edges have no cost increments. Consider a finite example run for this



(a) Priced Timed Automaton.

(b) Simulation of Priced Timed Automaton.

Fig. 4: CPU Priced Timed Automaton (a) and its simulation (b).

Priced Timed Automaton:

$$\pi_{PTA} = (\texttt{Idle}, x = 0, i = 0) \xrightarrow{2}_2 (\texttt{Idle}, x = 2, i = 0) \xrightarrow{\epsilon}_0 (\texttt{Work}, x = 0, i = 1)$$
$$\xrightarrow{3}_6 (\texttt{Work}, x = 3, i = 1) \xrightarrow{\epsilon}_0 (\texttt{Idle}, x = 0, i = 1) \xrightarrow{4}_4 (\texttt{Idle}, x = 5, i = 1)$$

The cost of π_{PTA} is thus $2 + 6 + 4 = 12$.

Hybrid Automata

To capture systems with more complex and rich dynamics, the formalism of Hybrid Automata [70] shall be used. Here, the evolution rates for clocks may not only be expressed with discrete variables, but can also depend on continuous variables including other clocks. Effectively, this allows capturing the dynamics with ordinary differential equations (ODEs).

Definition 8
A Hybrid Automaton (HA) is a tuple $(L, l_0, \mathcal{C}, \mathcal{X}, \Sigma, E, I, F)$ representing a Timed Automaton whose clock rates can now be set to be either constants or expressions that may depend on other variables. The delay function $F(d, v)$ provides a new valuation given delay $d \in \mathbb{R}_{\geq 0}$ and valuation v.

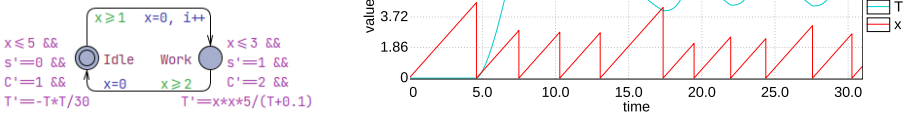
In the underlying semantics of a Priced Timed Automaton the *delay transition* is defined as follows:

- $(l, v) \xrightarrow{d} (l, v')$ iff $d \in \mathbb{R}_{\geq 0}$ and $v' = F(d, v)$ and $v' \models I(l)$

Unfortunately, the expressivity of Hybrid Automata comes at a cost. The decidability of different subclasses of Hybrid Automata has been intensely studied [6, 24, 70, 73, 94, 101, 110]. The most interesting to us is a subclass known as Rectangular Hybrid Automata (RHA) [71, 72]. RHA has two restrictions: (1) a variable must be reset of before its slope can change and (2) the two variables with different activities must not be compared. It was shown by [73] that relaxing any of the two restrictions renders RHA undecidable w.r.t. reachability and language emptiness problems, which proves that RHA resides on the border of what is decidable. Hence, reachability is also undecidable for the general class of Hybrid Automata.

Example 1.8 (Hybrid Automaton)
In case of a Hybrid Automaton (a) in Figure 5, we track the temperature of CPU which follows rich dynamics specified by ODEs. The temperature `T` decreases at a rate of `-T*T/30` when CPU is idling, and increases at a rate of `x*x*5/(T+0.1)` when CPU is processing a task. The simulation of (b) displays the evolution of CPU temperature (blue line) in degrees over time.

> In this simulation, the temperature stays under 10 degrees as the "work" is done quickly and there is plenty of time spent idling, which allows CPU to cool down.



(a) Hybrid Automaton.

(b) Simulation of Hybrid Automaton.

Fig. 5: CPU Hybrid Automaton (a) and its simulation (b).

Stochasticity

All of the models described above can exhibit stochastic behavior by employing race-based stochastic semantics. In a stochastic model, the nondeterministic choices of delays and transitions are replaced by stochastic ones[1]. The stochastic semantics has been studied for Stochastic Timed Automata [7], Stochastic Priced Timed Automata [45] and Stochastic Hybrid Automata [44]. It is possible to compose an arbitrary number of stochastic automata (components) into a network of stochastic automata where the components together act as a single system. The winning component at each step is decided according to the race – the component with the smallest delay, the winner, is the one that gets to perform its output action while the rest of the network may follow. The race-based stochastic semantics gives a basis for interpretation of the specification formalism and can be used to generate random, realistic simulations of the system. In fact, the graphs shown in Figures 1 (b), 3 (b), 4 (b) and 5 (b) showcase such stochastic simulations where both delays and transitions were chosen uniformly at random.

Along with the stochastic variants of Timed Automata and respective extensions, also reside timed variants of Markov Chains: Discrete Time Markov Chains (DTMC) [43] and Continuous Time Markov Chains (CTMC) [12] – both probabilistic models. Each state in DTMC has an explicitly specified probability distribution over the outgoing transitions which determine how likely each successor is to be picked. CTMC extends DTMC with exponential distributions of explicitly specified rates in each state which determine the probability of spending certain amount of time in that state.

[1] In Uppaal Stratego the standard choices are either uniform of exponential; however, any other distribution can be encoded.

1.2 Formal requirements

In addition to the model of the interest, model checking requires a formal requirement specification of the desired behavior (property) to validate the model against. Two classical examples are *reachability* and *safety* properties. Reachability property asks whether there exists a behavior that eventually satisfies the property, i.e. reaches the desired configuration. On the other hand, safety property requires that a certain condition is always satisfied (e.g. the temperature never exceeds a threshold). However, conclusive verification of safety properties cannot be done with non-exhaustive falsification techniques. Crucial to us, a safety property can be expressed as a reachability problem that upon satisfaction violates the safety property (e.g. it is possible to exhibit a behavior where the temperature exceeds a threshold).

More general behavioral properties can be expressed in different temporal logics. The two most relevant for this thesis, conventional families of temporal logics are linear-time (path-based) *Linear Temporal Logic* (LTL) [109] and branching-time (state-based) *Computation Tree Logic* (CTL) [40]. LTL views only a single successor (of behavior), i.e. a single trace, whereas CTL has a tree-like, branching structure and considers all alternative successors. Both logics allow to reason about infinite behavior of reactive systems and are a subset of CTL* [56]. Both LTL and CTL in their core support only discrete-time which suffices for the class of synchronous systems. To capture the real-time constraints and express properties of Timed Automata systems, *Timed* CTL (TCTL) [5] has been proposed, augmenting CTL with timing modalities. CTL has also been extended for probabilistic system with time and probabilities, resulting in Probabilistic CTL (PCTL) [66]. LTL has also been extended to continuous time and generalized to Metric Temporal Logic (MTL) [90]. Since full MTL over infinite traces is undecidable, an often preferred formalism is Metric Interval Temporal Logic (MITL) [9]. MITL is subset of MTL where the bounds of the operator time intervals are constrained to be either natural-valued or infinite.

1.3 Tools

There is a number of automated verification tools that have emerged as a result of the intensive research and improvement of the model-checking techniques. We now give a short summary of the closely related tools to the modelling and logic formalisms presented earlier.

Uppaal [17] is the prominent family of tools for design, specification and verification of real-time systems modelled as networks of Timed Automata that uses a subset of TCTL as the query language to define formal requirements. For verification of Timed Automata and TCTL formulae, the Kronos tool [27] can also be used; however, with the latest release in 2002, many of

the new and efficient methods are not present in KRONOS.

The implementation of Statistical Model Checking (SMC) is available in UPPAAL SMC branch that supports simulation of stochastic Timed Automata models (including Priced, Stopwatch and Hybrid Automata). The more complex dynamics of Hybrid Automata defined by ODEs are approximated during generation of simulation. A well-known alternative for evaluation of stochastic models is the symbolic probabilistic model checker PRISM [91] which supports a range of Markov Chain and Markov Decision Process models, such as DTMC and CTMC, and a wide range of temporal logics including LTL and PCTL. As of one of the more recent versions, PRISM 4.0 [92] also operates on *(Priced) Probabilistic Timed Automata* [62, 78, 93], that can be augmented with costs or rewards (similarly to Priced Timed Automata, but with probabilities).

For verification of networks of Priced Timed Automata, the tool UPPAAL CORA [20] exists, where resource-allocation problems are recast as optimal reachability problems. The algorithm of UPPAAL CORA can find optimal solutions, without providing anytime results[2]. Some of the more recent advancements in the methods for analysis of Priced Timed Automata gave rise to the tool TiaMo [26]. It only supports a subset of the syntax of Priced Timed Automata, but provides anytime solutions.

UPPAAL ECDAR [46] is a tool that can be used to check refinement relation between specifications expressed as Timed I/O Models. Behind the scenes, an engine of UPPAAL TIGA [16] is used, which in itself is a UPPAAL extension that supports the class of Timed Games and can synthesize strategies for those. The refinement between specifications is resolved in a 2-player game, where the inputs are controlled by the environment and the outputs by the component itself. There also exists an open-source spin-off tool Ecdar 2.x [1] which introduces new graphical interface and currently has two underlying engines with a constantly developed set of supported features.

1.4 Approaches

This section gives a summary of approaches that can be used to address the three problems classes – correctness, refinement and optimization of systems. It also highlights some other, relevant to the second part of this thesis, areas.

Testing

The most straightforward and widely used approach is that of testing. It allows one to identify most of the errors in the system rather quickly and cost-efficiently by executing a finite set of test-cases. In applications where the

[2]Intermediate (anytime) solutions are unfortunately not provided in the current distribution of UPPAAL CORA.

tolerance to errors is high, testing is a sufficient approach. However, discovery of more subtle bugs requires an increasing effort in test-case generation – often a manual process that involves creativity and discipline, and is prone to errors. Different testing techniques have been put through to improve the ability to detect bugs by e.g. generating corner-case test procedures as in boundary-value testing [112], generating test cases based on equivalence partitioning [21], or generating malformed inputs as in fuzz testing [87] to name a few. Moreover, a large number of automated testing and deployment tools exist today that help to reduce human involvement by process automation. Nevertheless, the major disadvantage of both functional (black-box) and structural (white-box) testing – lack of correctness guarantees – facilitates the need for different approaches.

Model checking

The approach of automated verification of system models against formalized requirements – known as *model checking* – has been established as a useful technique for providing correctness guarantees. Unfortunately, the state space grows exponentially in relation to the size of the system under analysis; this creates a problem known as *state-space explosion* – the most constraining obstacle that severely limits the applicability of model checking. A vast amount of effort has been directed over the last three decades to alleviate this problem, putting forward a large number of techniques all with the goal of reducing computational demands and improving scalability of verification methods.

Such techniques can be divided into four categories [41, 108]: reducing number of states to be explored (including symbolic methods), reducing memory requirements for storing these states, exploring abstractions of systems (over-approximation), exploring state-space in a parallel or distributed manner and exploring only part of the state-space at the cost of certainty (under-approximation).

In the setting of timed systems, different symbolic and discrete methods have been proposed. With discrete time, such algorithmic structures as Binary Decision Diagram [30] can be used to represent Boolean functions. This allows handling astronomically sized, but finitely representable, systems [32]. Difference Bound Matrix (DBM) [50] is the current state-of-the-art data structure used for symbolic state space representation in Uppaal, which is often referred to as *zone*. Clock Difference Diagrams (CDD) [13] is a BDD-like data structure that allows for saving space by a more efficient representation and manipulation of unions of zones than that provided by DBMs.

Some of the other prominent reduction techniques include *partial order reduction* [4, 22, 96] that reasons about redundancy of equivalent interleavings of independent concurrent events. *Extrapolation* abstraction techniques [14,

15, 119] reduce the number of states to explore which are semantically equivalent to already explored ones. *Symmetry reduction* [104] exploits user-defined symmetries in the models with the help of equivalence classes.

Probabilistic model checking

Some systems are designed in a way that imposes probabilities for certain events to occur. Examples include randomized algorithms, computer networks, concurrent protocols, and others. The traditional for model checking yes/no answer is then futile; instead, we are interested in the probability the target behavior can occur. Probabilistic model checking can be used to compute the probability of an event to occur from a probabilistic models like DTMC and CTMC and check whether that probability is below/above some desired threshold. The most prominent tool in this area is PRISM [91, 92]. Probabilistic model checking similarly suffers from the state space explosion as model checking, but also requires real-valued matrices and vectors, the calculations for which limit the size of tractable models even further.

Optimization and planning

Problems from temporal planning domain can be effectively expressed in *Planning Domain Definition Language* (PDDL) [3] specification language. Different PDDL extensions have improved the expressivity of the language over time, which includes probabilistic effects [124], temporal and numeric properties [57], and soft and strong constraints on plan trajectories [59]. A number of planner tools exist (e.g. FastDownward [68]) that implement classical planning algorithms, such as greedy best-first search with the FF heuristic for sub-optimal plans [75] and A* with LM-Cut for optimal plans [69].

Resource-allocation problems, such as optimization and planning problems, can be encoded also into Priced Timed Automata, effectively expressed as optimal reachability problems. It fact, PDDL can be translated [51] into Priced Timed Automata and analyzed with Uppaal Cora. The symbolic, exhaustive algorithm of Uppaal Cora performs an optimal reachability analysis of networks of Priced Timed Automata with the help of so-called *priced zones* [10] – an extension of standard, symbolic DBM-based zones. The algorithm is exhaustive and is guaranteed for find the most optimal solution.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [42] is a family of algorithms to find optimal solutions that has proven to be effective for a wide range of problems [29], particularly game playing and planning problems. The search operates by simulating the model and building the search tree while prioritizing more

promising regions based on observed rewards. MCTS has a number of useful characteristics: it (1) performs an incremental and *asymmetric* exploration of the state space, which helps to (a noticeable extent) avoid the state space explosion, (2) provides *anytime* solutions, and (3) is *aheuristic* in nature meaning it does not require any domain-specific knowledge.

MCTS makes choices based on past observations and addresses the *exploration* vs. *exploitation* dilemma as a multi-armed bandit problem in order to maximize the cumulative reward. More specifically, an optimal choice (of a bandit arm) can be done with the help of *upper confidence bound* (UCB1) [11] which expects a logarithmic growth of the regret uniformly over the number of choices. Proposition to use UCB as a mechanism for choice of successor in the model gave rise to now most popular *upper confidence bound for trees* (UCT) [88, 89] algorithm in the MCTS family.

Refinement verification

The problem of refinement checking is addressed by the specification theory of Timed I/O Automata [47]. Here, the contract-based relationship between decomposed systems is examined. The introduced notion of *conjunction* is used to express the intersection of requirements of components. *Structural composition*, similarly as in networks of timed systems, helps to combine specifications. The operation of *quotienting* is dual to composition and rather "subtracts" the behavior from a larger system, allowing incremental design. Finally, if one component *refines* another one, an equivalent system can be obtained by replacing the second component with the first.

Schedulability

Schedulability problems can also be solved with model-based approach. Here, the use of Stopwatch Automata allows us to encode preemption. In fact, the power of Stopwatch Automata is truly impressive: it was shown that the Stopwatch Automata is as expressive as Linear Hybrid Automata [34]. The decidability of hybrid systems was intensely studied: reachability analysis for Linear Hybrid Automata was shown to be semi-decidable [6]. The state-of-the-art algorithm for Stopwatch Automata analysis in UPPAAL is over-approximate as the underlying DBM data structure can encode difference between 2 variables only. This means the conclusive verification is only possible when no error is discovered. Otherwise, the result may be spurious.

Simulation

An alternative to exhaustive, but expensive symbolic methods are *simulation-based* approaches. They enjoy several significant advantages, mainly those

of being cheap, supporting larger class of models and requiring only a simulatable system, where samples can be drawn according to the distribution defined by the system. The most renowned simulation-based method in the area of formal verification is Statistical Model Checking (SMC) [118, 123].

The core idea of SMC is to monitor a number of random, independent samples (traces), produced by simulation of a stochastic model, and use statistical methods (e.g. hypothesis testing) to estimate the probability that a random run will satisfy a given property. Hence, SMC can be used for verification of both qualitative properties, e.g. by estimating whether the probability is larger than zero for a yes/no answer, and quantitative properties, such as estimation of the probability being either below or above a certain threshold. The probability estimate comes as an interval of a certain precision with a certain level of confidence which requires more independent sample traces for higher precision and higher confidence. SMC is under-approximate in nature and cannot provide correctness guarantees; nonetheless, SMC can improve or weaken our confidence in the hypothesis and all together can be viewed as a compromise between traditional model checking and testing techniques. Due to simulation-based methods being much less time and memory demanding, they are often the only option for large models. Moreover, for models like Stopwatch Automata, where the existing symbolic reachability analysis is over-approximate, SMC can be used to generate concrete witnesses of property violation.

Importance sampling and splitting

From here we use the term of a *rare event*. A rare event is an event that has an extremely low probability of occurring during a random simulation of a system with respect to the stochastic semantics. The probability of a rare event can reside in the range $[0, 10^{-100}]$[3] or even smaller. In other words, rare events represent part of the state space that is either/both (1) very small in relation to the rest of the state space and/or (2) is very difficult to reach when exploring the model. Note that a rare event can equally be used in the context of any of the three challenges brought up earlier: rare error in the system, rare refinement relation violation or a rare optimal schedule.

In practice, we are often interested in applications with subtle and rare events which are hard to find. In those cases, SMC cannot estimate the probability reliably. This happens as the relative error (variance) of the samples grows inversely with the rarity of the searched event. One of the techniques to reduce the variance of simulations is known as *importance sampling* [82, 113]. It aims to increase the probability of reaching a rare event so that more of the "important" traces can be obtained, allowing to draw statistically significant conclusions faster. This is achieved by altering the

[3] In Paper D we encounter rare events with the probability smaller than 10^{-55}.

probability distributions of the system under test; however, the obtained results then must be changed to compensate for introduced alterations and to obtain an unbiased estimate. There exist different ways to implement the biasing schemes, all with the goal of increasing the rare event probability in comparison to the unbiased density.

Importance splitting [114] is an alternative variance reduction method. It reformulates the problem by partitioning a rare goal into a number of not-so-rare sub-goals. The observation is that while reaching the main goal from a starting configuration is difficult, reaching each next sub-goal from the previous one is much easier. The probability estimate is computed as a product of probabilities to reach sub-goals, and hence requires a much smaller number of simulations for the same statistical confidence. Both importance sampling and importance splitting techniques have also been implemented for Stochastic Timed Automata and integrated into in Uppaal SMC [77, 98].

Counterexample detection

A different path to indirectly cope with the state space explosion is the realization that some of the formal requirements can be violated as soon as a single witness of target behavior – a *counterexample* – is found. If such counterexample is found early in the search, no more effort needs to be spend on exploring the rest of the arbitrary large state space. Such techniques, that rely on quick and efficient violation of the property, can be grouped under the name of *counterexample detection* or *falsification* techniques. One subtle example is *random depth first search* (RDFS). It replaces deterministic nature of the *depth first search* (DFS) model checking algorithm with a uniformly distributed, random choice of a successor. With enough luck, RDFS can discover counterexamples quickly [52] in systems where other symbolic methods, even such as DFS, would be intractable.

While symbolic RDFS in Uppaal is complete due to its exhaustive nature, some counterexample detection techniques give up on the requirement of completeness and explore only part of the state space in hope to find counterexamples even faster. In the context of under-approximate simulation-based techniques, one may naturally want to exploit already existing method of SMC. While the stochastic semantics allows for a model to mimic the behavior of a real system, SMC may not be a productive approach for violation discovery as it primarily exercises the most frequent behavior of the system. This means that the SMC simulations are likely to be efficient at detecting frequently occurring errors caused by the common usage of the system, whereas very subtle and rare-to-occur bugs are likely to be missed. Performing multiple SMC simulations will eventually discover even counterexamples of an extremely low rarity; however, the computational time demands might not be better, if not worse, than that of the exhaustive model-checking. Therefore,

counterexample detection methods require different heuristics in order to be efficient.

2 Randomization in Model Checking

An interesting to us type of algorithms in computer science is randomized algorithms which use some *randomness* as part of the algorithm logic, typically with the aim of reducing the running time. Some versions of the sorting algorithm Quicksort [74] are a good example. Quicksort operates by separating the input sequence into two partitions based on whether the numbers are smaller or larger than some selected "pivot" value. In early version of Quicksort, the pivot value was selected deterministically, causing a worst-case performance for already sorted sequences that are a common use-case [35]. Replacing deterministic selection by a pseudorandomly generated choice was an easy way to improve performance.

Employing *randomness* is also a promising direction in the model checking algorithms. Some or all of the choices, such as choice of delays or transitions in models, can be randomized to obtain efficient counterexample detection methods. We now delve into a more in-depth discussion about the role of randomness in the field of model checking.

2.1 Related work

We want to start this discussion by asking the reader to think about the following. How much can randomness affect the performance of model checking methods? Even if we only introduce a small change e.g. by replacing a deterministic search order with a random one? To provide insights into how randomness affects the performance of DFS, the study [53] examined the default search order of various path-sensitive error-detection techniques. The authors believe that the reported performance benefits of different techniques can in some cases be caused by the default search order rather than the techniques themselves. In fact, the usage of randomness in DFS can provide dramatic performance improvements [52] of up to several orders of magnitude in multi-threaded Java programs, showing that random variations in the search order can produce huge variations in the results.

First application of randomness

One of the first steps in applying randomness were made by [67, 102, 115, 121, 122] to validate complex systems such as communication protocols. The authors argue that for large systems exhaustive reachability analysis is not effective due to astronomically large state spaces. Instead, the errors are

searched by means of *random walks*, i.e. randomized, independent simulations. In its core form, a random walk is a simulation of a graph-like model where successors are chosen uniformly at random.

Randomness in LTL model checking

The random walk has been formalized and extended to the setting of LTL model checking by [65]. Their method is to explore an intersection of a model with the Büchi Automaton [31] that represents a negated property. The target of the exploration are so-called *lassos* – a prefix followed by an elementary cycle. If the accepting state is present in the cycle portion then the lasso is called *accepting* and represents a counterexample to the searched property. The performed random walks draw successors uniformly at random.

A number of alternative sampling strategies are examined [95] in hope to improve the efficiency of the search and reduce the amount of random walks necessary to falsify the property. Furthermore, memory-efficient variants of those strategies that store a small, finite number of *tokens* (states) are studied. The alternative strategies in general show an improved performance and decreased memory consumption for tokenized variants. However, authors note that for all of the strategies it is possible to construct difficult models and advocate for running multiple strategies in parallel. Given that the counterexample exists, the running time will then be that of the fastest strategy. Regardless of the strategy, one persistent issue is rarity of counterexamples. In such case, a very large number of random walks must be made to ensure high coverage of lassos. To aid the situation, an efficient algorithm for counting and generating lassos uniformly was proposed [107], but only for a sub-class of directed graphs, known as reducible flow graphs.

Randomness in guided search

In model checking some techniques focus on guiding the search towards more promising areas of the state space to improve counterexample detection. In *guided search*, the states are ranked in a queue by a heuristic according to some knowledge about the model or the property. During the exploration, the search algorithm prioritizes the states that were estimated to lead to the counterexample faster over the rest of the states. Guided search usually performs better than DFS or BFS, provided the ranking heuristics are accurate, and provides generally shorter traces to a counterexample, simplifying debugging process.

The benefits of randomness have also been explored in the guided search where a successor is chosen randomly from n best states in the priority queue [80]. The authors claim that multiple parallel guided searches increase the average expected time to find an error, but instead decrease the minimum

expected time in the same setting; however, the results are not clearly in favor of employing randomness. A convincing benefit of randomness was shown by completely shuffling the states with equal heuristic ranking [116]. The authors stress the importance of tailoring heuristic well to both the model and the property as otherwise the efficiency of RDFS is better than that of randomized guided search.

Randomization in Timed Automata

Random walks are often memory-less by nature and each next random walk is independent of a previous one. Because of this, some of the states can be explored multiple times. Such *redundancy* can be prohibitive and is one of the main drawbacks of random walks. The amount of redundancy usually depends on the model structure and exact behavior of the algorithm. The issue gets worse in systems where random walks frequently end up in isolated parts of the state space that contain no target behavior. A common way to address this issue is by periodically restarting random walks to avoid blocking (being stuck) in some fraction of the state space. In the setting of Timed Automata, a *Deep Random Search* (DRS) has been proposed [64]. DRS is an exhaustive, symbolic Las Vegas algorithm[4] where random walks traverse the state space up to a specified cut-off depth. The random walks are restarted from other, previously explored, states which helps to minimize redundancy and reach deep states. Moreover, with its *iterative deepening*, DRS facilities a search for shortest trace that is optimal up to a value of the *increment* for a cut-off depth.

Decreasing memory requirements

While hardware became cheap and more accessible over time, memory is still often a prohibitive factor when applying model checking for larger systems. To address that, randomness is also used specifically to harness memory requirements of algorithms. In [28], randomness helps to decide which visited states to store and which ones to discard in the context of automata based LTL model checking: a noticeable memory reduction is achieved at a cost of only minor time overhead. *Resource-aware* algorithms have been introduced [2] to conduct, among other options, a memory-aware exploration. Their *Uniform Random Search* (URS) and *Simplified Deep Random Search* (SDRS) – both memory-aware – have been shown to explore up to 40% more states than breadth first search (BFS) that terminates after running out of memory. In [120], a randomized variant of BFS (RBFS) with the fixed amount of memory is examined. In essence, their algorithm can be viewed as an array

[4] A Las Vegas algorithm employs randomness and is guaranteed to terminate and produce a correct result (or report a failure).

of random walks with memory. Once the memory is filled, the algorithm proceeds at lower speed instead of terminating completely. RBFS is reported to be 100% slower in average, but explores 30% more state space in average in comparison to deterministic exploration that exhausts the memory and terminates.

In another take on Timed Automata, a number of efficient storing strategies were explored to reduce the number of stored states [19]. One of such strategies requires computation of a *covering set* – a set of edges with the property that each cycle in the state space contains at least one edge for that set. Randomness is used both as a separate storing strategy, but also as one of the heuristics to compute the covering set – an otherwise NP-complete problem. The performance of randomness-employing methods is overall on-par with the rest of the techniques and varies from model to model.

More randomness

Furthermore, randomness is used in a number of other model checking related studies which include, but are not limited to state [60] and state-space caching [58] techniques, HSF-SPIN model checker for safety and LTL-specified liveness properties [54], analysis of Java programs using structural heuristics [63], finding errors in very large concurrent reactive systems [61], validating programs based on guiding statistical sampling of inputs [117], cross-entropy based testing [36], uniform random sampling of traces in very large models [49, 106], coverage-biased randomised exploration in composed automata models [48], and parallel random exhaustive hardware in the loop simulation [100]. In all of these applications, randomization has contributed to an overall performance improvement of a method.

2.2 Our take

Based on the related work presented we believe that wisely used randomization can have a huge potential on the performance of model checking w.r.t. state space explosion problem. To the best of our knowledge, there were little to no attempts to apply randomized checking for various extensions of Timed Automata models in the context of a memory-less, cheap, lightweight and efficient random walks. We want to investigate this promising direction and to work with concrete semantics which allows us to avoid expensive computations of symbolic abstractions.

Hypothesis: *We believe that by employing randomness we will be able to develop lightweight, efficient and highly scalable methods for a quick checking of correctness, refinement and optimization problems in timed systems.*

3 Thesis Summary

We now give a overview of each paper and their respective contributions.

Paper A: Randomized Refinement Checking of Timed I/O Automata

Abstract: To combat the *state-space explosion* problem and ease system development, we present a new refinement checking (falsification) method for Timed I/O Automata based on random walks. Our memory-less heuristics *Random Enabled Transition* (RET) and *Random Channel First* (RCF) provide efficient and highly scalable methods for counterexample detection. Both RET and RCF operate on concrete states and are relieved from expensive computations of symbolic abstractions. We compare the most promising variants of RET and RCF heuristics to existing symbolic refinement verification of the ECDAR tool. The results show that as the size of the system increases our heuristics are significantly less prone to exponential increase of time required by ECDAR to detect violations: in very large systems both "wide" and "narrow" violations are found up to 600 times faster and for extremely large systems when ECDAR timeouts, our heuristics are successful in finding violations.

Contribution 1
We designed and implemented a randomized refinement checking algorithm, i.e. an algorithm for efficient falsification of the refinement relation between Timed I/O Automata. We compared two proposed heuristics of our algorithm, and their respective variants, to state-of-the-art alternatives for refinement checking – UPPAAL ECDAR and a Java implementation of SMC for refinement reformulated as a reachability problem. Our methods have shown a significant improvement in performance for all the experiments: up to 5 order of magnitude faster than SMC and up to 3 orders of magnitude faster than UPPAAL ECDAR.

Publication History: This paper [84] was accepted and presented at *6th International Symposium on Dependable Software Engineering. Theories, Tools, and Applications* (SETTA 2020) and published in proceedings of SETTA 2020 LNPSE volume 12153, pages 70-88.

Paper B: Randomized Reachability Analysis in UPPAAL: Fast Error Detection in Timed Systems

Abstract: Randomized reachability analysis is an efficient method for detection of safety violations. Due to the under-approximate nature of the method, it excels at quick falsification of models and can greatly improve the model-based development process: using lightweight randomized methods early in the development for the discovery of bugs, followed by expensive symbolic verification only at the very end. We show the scalability of our method on a number of Timed Automata and Stopwatch Automata models of varying sizes and origin. Among them, we revisit the schedulability problem from the Herschel-Planck industrial case study, where our new method finds the deadline violation three orders of magnitude faster: some cases could previously be analyzed by statistical model checking (SMC) in 23 hours and can now be checked in 23 seconds. Moreover, a deadline violation is discovered in a number of cases that where previously intractable. We have implemented the Randomized reachability analysis – and made it available – in the tool Uppaal. Finally we provide an evaluation of the strengths and weaknesses of Random reachability analysis exploring exactly which types of model features hamper method's efficiency.

Contribution 2
We developed and implemented a randomized reachability analysis that performs an under-approximate, but efficient falsification of reachability problems for Timed and Stopwatch Automata models. The method and its respective heuristics were extensively tested on a number of academic- and industrial-sized models, including Herschel-Planck satellite, Java bytecode systems and ARINC avionics systems. The results were truly impressive: the performance of the randomized reachability analysis algorithm managed to find requirement violations up to three orders of magnitude faster than other state-of-the-art alternatives.

Contribution 3
The randomized reachability analysis has been implemented and is now available in the tool Uppaal for falsification of reachability queries in Timed and Stopwatch Automata models.

Publication History: The first version [85] was accepted and presented at *26th International Conference on Formal Methods for Industrial Critical Systems* (FMICS 2021) and published in proceedings of FMICS 2021 LNCS volume 12863, pages 149-166. The present, extended version [86] was accepted and published at *International Journal on Software Tools for Technology Transfer* (STTT 2022) volume 24 issue 6 (pages 1025-1042), devoted to FMICS 2021.

Paper C: Monte Carlo Tree Search for Priced Timed Automata

Abstract: Priced timed automata (PTA) were introduced in the early 2000s to allow for generic modelling of resource-consumption problems for systems with real-time constraints. Optimal schedules for allocation of resources may here be recast as optimal reachability problems. In the setting of PTA this problem has been shown decidable and efficient symbolic reachability algorithms have been developed. Moreover, PTA has been successfully applied in a variety of applications. Still, we believe that using techniques from the planning community may provide further improvements. Thus, in this paper we consider exploiting Monte Carlo Tree Search (MCTS), adapting it to problems formulated as PTA reachability problems. We evaluate our approach on a large benchmark set of PTAs modelling either Task graph or Job-shop scheduling problems. We discuss and implement different complete and incomplete exploration policies and study their performance on the benchmark. In addition, we experiment with both well-established and our novel MTCS-based optimizations of PTA and study their impact. We compare our method to the existing symbolic optimal reachability engines for PTAs and demonstrate that our method (1) finds near-optimal plans, and (2) can construct plans for problems infeasible to solve with existing symbolic planners for PTA.

Contribution 4
We adapted Monte Carlo Tree Search (MCTS) algorithm for analysis of Priced Timed Automata in search for optimal solutions. We implemented discrete-time MCTS in Uppaal, including all developed tree unfolding strategies and enhancements. We compared our method to other existing state-of-the-art methods and tools, including classical algorithms from the planning domain. Results demonstrate the superiority of MCTS in providing (near-) optimal solutions where other exhaustive methods do not terminate.

Publication History: This paper [79] was accepted and presented at *19th International Conference on Quantitative Evaluation of Systems* (QEST 2022) and published in proceedings of QEST 2022 LNCS volume 13479, pages 381-398.

Paper D: Usage-aware Falsification for Cyber-Physical Systems

Abstract: Verification of cyber-physical systems (CPS) is a challenging task. A considerable effort has been invested to develop pragmatic methods, such as falsification testing, which facilitate generation of inputs that lead to the violation of the CPS requirements. The resulting counter-examples are used to locate and explain faults and debug the system. However, CPS rarely operate in fully unconstrained environments and not all counter-examples have the same value – a fault resulting from a common usage of the system has more impact than a fault that is triggered by an esoteric input sequence. This aspect is neglected by the existing falsification testing techniques. We propose a new falsification testing methodology that is aware of the system's expected usage. Given a user profile model in the form of a stochastic hybrid automaton, an executable black-box implementation of the CPS and its formalized requirements, we provide a test generation method that (1) uses efficient randomized methods to generate multiple violating traces, and (2) estimates the probability of each counterexample, thus providing their ranking to the engineer.

Contribution 5
We extended randomized reachability analysis in Uppaal to support Hybrid Automata model where clocks can evolve according to ODEs. The ODEs are solved by a Runge-Kutta approximation method. Moreover, we extended the support to simulation queries that can now be used with randomized exploration algorithm and outputs a simulation trace of desired quantitative aspects of a model.

Contribution 6
We proposed and developed an efficient framework for falsification of black-box cyber-physical systems. To the best of our knowledge, in the setting of falsification-based testing this is the first framework that supports reasoning about usage-awareness of system under test, generates multiple counterexamples, and estimates the probability of each counterexample. The probability is estimated by a combination of randomized and statistical simulation methods w.r.t. the stochastic semantics of the environment model. The estimated probability allowed us to rank the counterexamples, thus facilitating the debugging process.

Publication History: The paper was submitted to *14th International Conference on Cyber-Physical Systems* (ICCPS 2023), but was rejected. The paper version presented in this thesis has been revised according to the reviewers' feedback. Currently, a new version of the paper is being prepared to be submitted to another conference.

References

[1] Ecdar2.0. [Online]. Available: https://www.ecdar.net/

[2] N. Abed, S. Tripakis, and J.-M. Vincent, "Resource-Aware Verification Using Randomized Exploration of Large State Spaces," in *Model Checking Software*, K. Havelund, R. Majumdar, and J. Palsberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 214–231.

[3] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson *et al.*, "Pddl| the planning domain definition language," *Technical Report, Tech. Rep.*, 1998.

[4] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-Order Reduction in Symbolic State Space Exploration," in *Computer Aided Verification*, O. Grumberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 340–351.

[5] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, no. 1, pp. 2–34, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540183710242

[6] R. Alur, C. Courcoubetis, N. Halbwachs, T. 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, hybrid Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759400202T

[7] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for probabilistic real-time systems," in *Automata, Languages and Programming*, J. L. Albert, B. Monien, and M. R. Artalejo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 115–126.

[8] R. Alur and D. Dill, "The theory of timed automata," in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Springer, 1992, pp. 45–73.

[9] R. Alur, T. Feder, and T. A. Henzinger, "The benefits of relaxing punctuality," *Journal of the ACM (JACM)*, vol. 43, no. 1, pp. 116–146, 1996.

[10] R. Alur, S. La Torre, and G. J. Pappas, "Optimal Paths in Weighted Timed Automata," in *HSCC 2001*, M. D. Di Benedetto and A. Sangiovanni-Vincentelli, Eds. Springer, 2001, pp. 49–62.

[11] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

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

[13] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi, "Efficient Timed Reachability Analysis Using Clock Difference Diagrams," in *Computer Aided Verification*, N. Halbwachs and D. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 341–353.

[14] G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen, "Static Guard Analysis in Timed Automata Verification," in *TACAS 2003*, H. Garavel and J. Hatcliff, Eds. Springer, 2003, pp. 254–270.

[15] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek, "Lower and Upper Bounds in Zone Based Abstractions of Timed Automata," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 312–326.

[16] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Time for Playing Games!" in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 121–125.

[17] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer, 2004, pp. 200–236.

[18] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager, "Minimum-Cost Reachability for Priced Time Automata," in *HSCC 2001*, M. D. Di Benedetto and A. Sangiovanni-Vincentelli, Eds. Springer, 2001, pp. 147–161. [Online]. Available: https://doi.org/10.1007/3-540-45351-2_15

[19] G. Behrmann, K. G. Larsen, and R. Pelánek, "To Store or Not to Store," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 433–445.

[20] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Priced Timed Automata: Algorithms and Applications," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer, 2005, pp. 162–182.

[21] A. Bhat and S. M. K. Quadri, "Equivalence class partitioning and boundary value analysis - a review," in *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, 2015, pp. 1557–1562.

[22] F. M. Bønneland, P. G. Jensen, K. G. Larsen, M. Muñiz, and J. Srba, "Start pruning when time gets urgent: Partial order reduction for timed systems," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 527–546.

[23] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 836–850. [Online]. Available: https://doi.org/10.1145/3477132.3483540

[24] A. Bouajjani and R. Robbana, "Verifying ω-regular properties for a subclass of linear hybrid systems," in *Computer Aided Verification*, P. Wolper, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 437–450.

[25] P. Bouyer, T. Brihaye, V. Bruyère, and J. Raskin, "On the optimal reachability problem of weighted timed automata," *Formal Methods Syst. Des.*, vol. 31, no. 2, pp. 135–175, 2007.

[26] P. Bouyer, M. Colange, and N. Markey, "Symbolic Optimal Reachability in Weighted Timed Automata," in *CAV 2016*, S. Chaudhuri and A. Farzan, Eds. Springer, 2016, pp. 513–530.

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

[28] L. Brim, I. Černá, and M. Nečesal, "Randomization Helps in LTL Model Checking," in *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*, L. de Alfaro and S. Gilmore, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 105–119.

[29] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

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

[31] J. R. Büchi, *On a Decision Method in Restricted Second Order Arithmetic*. New York, NY: Springer New York, 1990, pp. 425–435. [Online]. Available: https://doi.org/10.1007/978-1-4613-8928-6_23

[32] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 1020 States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[33] A. Burns, *Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach*. USA: Prentice-Hall, Inc., 1995, p. 225–248.

[34] F. Cassez and K. Larsen, "The impressive power of stopwatches," in *CONCUR 2000 — Concurrency Theory*, C. Palamidessi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 138–152.

[35] B. Chandramouli and J. Goldstein, "Patience is a virtue: Revisiting merge and sort on modern processors," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 731–742. [Online]. Available: https://doi.org/10.1145/2588555.2593662

[36] H. Chockler, E. Farchi, B. Godlin, and S. Novikov, "Cross-entropy based testing," in *Formal Methods in Computer Aided Design (FMCAD'07)*, 2007, pp. 101–108.

[37] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic

model checking," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364.

[38] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, p. 244–263, apr 1986. [Online]. Available: https://doi.org/10.1145/5397.5399

[39] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*. Berlin, Heidelberg: Springer-Verlag, 1981, p. 52–71.

[40] ——, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, D. Kozen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71.

[41] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8

[42] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83.

[43] C. Courcoubetis and M. Yannakakis, "Verifying temporal properties of finite-state probabilistic programs," in *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, ser. SFCS '88. USA: IEEE Computer Society, 1988, p. 338–345. [Online]. Available: https://doi.org/10.1109/SFCS.1988.21950

[44] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikuč ionis, D. B. Poulsen, and S. Sedwards, "Statistical model checking for stochastic hybrid systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 92, pp. 122–136, aug 2012.

[45] 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*, U. Fahrenberg and S. Tripakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–96.

[46] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wąsowski, "Ecdar: An environment for compositional design and analysis of real time systems," in *Automated Technology for Verification and Analysis*, A. Bouajjani and W.-N. Chin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 365–370.

[47] 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: Association for Computing Machinery, 2010, p. 91–100.

[48] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet, "Coverage-biased random exploration of large models and application to testing," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 73–93, 2012.

[49] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, and S. Peyronnet, "Uniform random sampling of traces in very large models," in *Proceedings of the 1st International Workshop on Random Testing*, ser. RT '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 10–19. [Online]. Available: https://doi.org/10.1145/1145735.1145738

[50] D. L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," in *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 197–212.

[51] H. Dirks, "Finding Optimal Plans for Domains with Restricted Continuous Effects with UPPAAL CORA," ser. ICAPS 2005. American Association for Artificial Intelligence, 2005.

[52] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 3–12.

[53] M. B. Dwyer, S. Person, and S. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 92–104. [Online]. Available: https://doi.org/10.1145/1181775.1181787

[54] S. Edelkamp, A. L. Lafuente, and S. Leue, "Directed explicit model checking with hsf-spin," in *Model Checking Software*, M. Dwyer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 57–79.

[55] 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.

[56] E. A. Emerson and J. Y. Halpern, ""sometimes" and "not never" revisited: On branching versus linear time temporal logic," *J. ACM*, vol. 33, no. 1, p. 151–178, jan 1986. [Online]. Available: https://doi.org/10.1145/4904.4999

[57] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[58] J. Geldenhuys, "State caching reconsidered," in *Model Checking Software*, S. Graf and L. Mounier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 23–38.

[59] A. Gerevini and D. Long, "Plan constraints and preferences in pddl3," Technical Report 2005-08-07, Department of Electronics for Automation ..., Tech. Rep., 2005.

[60] P. Godefroid, G. J. Holzmann, and D. Pirottin, "State-space caching revisited," *Formal Methods in System Design*, vol. 7, no. 3, pp. 227–241, 1995.

[61] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," in *Tools and Algorithms for the Construction and Analysis of Systems*, J.-P. Katoen and P. Stevens, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 266–280.

[62] H. Gregersen and H. E. Jensen, "Design of real-time probabilistic logic," Master's thesis, Aalborg University, 1995.

[63] A. Groce and W. Visser, "Model checking java programs using structural heuristics," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 12–21. [Online]. Available: https://doi.org/10.1145/566172.566175

[64] R. Grosu, X. Huang, S. A. Smolka, W. Tan, and S. Tripakis, "Deep Random Search for Efficient Model Checking of Timed Automata," in *Composition of Embedded Systems. Scientific and Industrial Issues*, F. Kordon and O. Sokolsky, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 111–124.

[65] R. Grosu and S. A. Smolka, "Monte Carlo Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Halbwachs and L. D. Zuck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 271–286.

[66] 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.

[67] P. Haslum, "Model checking by random walk," in *Proceedings of the ECSEL Workshop (CCSSE)*, 1999.

[68] M. Helmert, "The fast downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[69] M. Helmert and C. Domshlak, "Landmarks, critical paths and abstractions: what's the difference anyway?" in *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[70] T. A. Henzinger, *The Theory of Hybrid Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 265–292. [Online]. Available: https://doi.org/10.1007/978-3-642-59615-5_13

[71] T. A. Henzinger and P. W. Kopke, "State equivalences for rectangular hybrid automata," in *CONCUR '96: Concurrency Theory*, U. Montanari and V. Sassone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 530–545.

[72] ——, "Discrete-time control for rectangular hybrid automata," in *Automata, Languages and Programming*, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 582–593.

[73] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, 1995, pp. 373–382.

[74] C. A. R. Hoare, "Algorithm 64: quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, 1961.

[75] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[76] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[77] C. Jegourel, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards, "Importance sampling for stochastic timed automata," in *Dependable Software Engineering: Theories, Tools, and Applications*, M. Fränzle, D. Kapur, and N. Zhan, Eds. Cham: Springer International Publishing, 2016, pp. 163–178.

[78] H. Jensen, "Model checking probabilistic real time systems," 01 2002.

[79] P. G. Jensen, A. Kiviriga, K. Guldstrand Larsen, U. Nyman, A. Mijačika, and J. Høiriis Mortensen, "Monte carlo tree search for priced timed automata," in *Quantitative Evaluation of Systems*, E. Ábrahám and M. Paolieri, Eds. Cham: Springer International Publishing, 2022, pp. 381–398.

[80] M. Jones and E. Mercer, "Explicit state model checking with hopper," in *Model Checking Software*, S. Graf and L. Mounier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 146–150.

[81] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 01 1986. [Online]. Available: https://doi.org/10.1093/comjnl/29.5.390

[82] H. Kahn, *Use of different Monte Carlo sampling techniques*. Rand Corporation, 1955.

[83] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, "Timed i/o automata: a mathematical framework for modeling and analyzing real-time systems," in *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, 2003, pp. 166–177.

[84] A. Kiviriga, K. G. Larsen, and U. Nyman, "Randomized Refinement Checking of Timed I/O Automata," in *Dependable Software Engineering. Theories, Tools, and Applications*, J. Pang and L. Zhang, Eds. Cham: Springer International Publishing, 2020, pp. 70–88.

[85] ——, "Randomized Reachability Analysis in Uppaal: Fast Error Detection in Timed Systems," in *FMICS 2021*, A. Lluch Lafuente and A. Mavridou, Eds. Springer, 2021, pp. 149–166.

[86] ——, "Randomized reachability analysis in UPPAAL: fast error detection in timed systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 24, no. 6, pp. 1025–1042, 2022. [Online]. Available: https://doi.org/10.1007/s10009-022-00681-z

[87] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[88] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[89] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved monte-carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[90] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[91] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic symbolic model checker," in *Computer Performance Evaluation: Modelling Techniques and Tools*, T. Field, P. G. Harrison, J. Bradley, and U. Harder, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 200–204.

[92] ——, "Prism 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 585–591.

[93] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic verification of real-time systems with discrete probability distributions," *Theoretical Computer Science*, vol. 282, no. 1, pp. 101–150, 2002, real-Time and Probabilistic Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397501000469

[94] G. Lafferriere, G. J. Pappas, and S. Yovine, "A new class of decidable hybrid systems," in *Hybrid Systems: Computation and Control*, F. W. Vaandrager and J. H. van Schuppen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 137–151.

[95] K. Larsen, D. Peled, and S. Sedwards, "Memory-Efficient Tactics for Randomized LTL Model Checking," in *Verified Software. Theories, Tools, and Experiments*, A. Paskevich and T. Wies, Eds. Cham: Springer International Publishing, 2017, pp. 152–169.

[96] K. G. Larsen, M. Mikučionis, M. Muñiz, and J. Srba, "Urgent partial order reduction for extended timed automata," in *Automated Technology for Verification and Analysis*, D. V. Hung and O. Sokolsky, Eds. Cham: Springer International Publishing, 2020, pp. 179–195.

[97] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 134–152, Dec 1997. [Online]. Available: https://doi.org/10.1007/s100090050010

[98] K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, "Importance splitting in uppaal," in *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part III*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 13703. Springer, 2022, pp. 433–447. [Online]. Available: https://doi.org/10.1007/978-3-031-19759-8_26

[99] N. A. Lynch and M. R. Tuttle, *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[100] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci, "Anytime system level verification via parallel random exhaustive hardware in the loop simulation," *Microprocessors and Microsystems*, vol. 41, pp. 12–28, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933115002045

[101] J. McManis and P. Varaiya, "Suspension automata: A decidable class of hybrid automata," in *Computer Aided Verification*, D. L. Dill, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 105–117.

[102] M. Mihail and C. H. Papadimitriou, "On the random walk method for protocol testing," in *Computer Aided Verification*, D. L. Dill, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 132–141.

[103] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, p. 66–73, mar 2015. [Online]. Available: https://doi.org/10.1145/2699417

[104] C. Norris IP and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1, pp. 41–75, 1996.

[105] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 13–25. [Online]. Available: https://doi.org/10.1145/3209108.3209109

[106] J. Oudinet, "Uniform random walks in very large models," in *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ser. RT '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 26–29. [Online]. Available: https://doi.org/10.1145/1292414.1292422

[107] J. Oudinet, A. Denise, M.-C. Gaudel, R. Lassaigne, and S. Peyronnet, "Uniform Monte-Carlo Model Checking," in *Fundamental Approaches to Software Engineering*, D. Giannakopoulou and F. Orejas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 127–140.

[108] R. Pelánek, "Fighting state space explosion: Review and evaluation," in *Formal Methods for Industrial Critical Systems*, D. Cofer and A. Fantechi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 37–52.

[109] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.

[110] A. Puri and P. Varaiya, "Decidability of hybrid systems with rectangular differential inclusions," in *Computer Aided Verification*, D. L. Dill, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 95–104.

[111] 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.

[112] M. Ramachandran, "Testing software components using boundary value analysis," in *2003 Proceedings 29th Euromicro Conference*, 2003, pp. 94–98.

[113] G. Rubino and B. Tuffin, *Rare event simulation using Monte Carlo methods*. John Wiley & Sons, 2009.

[114] ——, *Rare event simulation using Monte Carlo methods*. John Wiley & Sons, 2009.

[115] H. Rudin, "Protocol development success stories: Part i," in *Protocol Specification, Testing and Verification, XII*, ser. IFIP Transactions C: Communication Systems, R. LINN and M. UYAR, Eds. Amsterdam: Elsevier, 1992, pp. 149–160. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444898746500167

[116] N. Rungta and E. G. Mercer, "Generating counter-examples through randomized guided search," in *Model Checking Software*, D. Bošnački and S. Edelkamp, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 39–57.

[117] S. Sankaranarayanan, R. M. Chang, G. Jiang, and F. Ivančić, "State space exploration using feedback constraint generation and monte-carlo sampling," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 321–330. [Online]. Available: https://doi.org/10.1145/1287624.1287670

[118] K. Sen, M. Viswanathan, and G. Agha, "Statistical Model Checking of Black-Box Probabilistic Systems," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 202–215.

[119] S. Tripakis and C. Courcoubetis, "Extending promela and spin for real time," in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 329–348.

[120] E. Tronci, G. Della Penna, B. Intrigila, and M. Zilli, "A probabilistic approach to automatic verification of concurrent systems," in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, 2001, pp. 317–324.

[121] C. H. West, "Protocol validation in complex systems," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 303–312. [Online]. Available: https://doi.org/10.1145/75246.75276

[122] C. West, "Protocol validation by random state exploration," in *Protocol Specification, Testing, and Verification*, 1987, pp. 233–242.

[123] H. L. S. Younes and R. G. Simmons, "Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 223–235.

[124] H. L. Younes and M. L. Littman, "Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects," *Techn. Rep. CMU-CS-04-162*, vol. 2, p. 99, 2004.

Part II

Papers

Paper A

Randomized Refinement Checking of Timed I/O Automata

Andrej Kiviriga, Kim Guldstrand Larsen, Ulrik Nyman

The paper has been published in the
Proceedings of Dependable Software Engineering. Theories, Tools, and Applications SETTA 2020, LNPSE Vol. 12153, pp. 70-88, 2020.

The layout has been revised.

Abstract

To combat the state-space explosion problem and ease system development, we present a new refinement checking (falsification) method for Timed I/O Automata based on random walks. Our memory-less heuristics Random Enabled Transition (RET) and Random Channel First (RCF) provide efficient and highly scalable methods for counterexample detection. Both RET and RCF operate on concrete states and are relieved from expensive computations of symbolic abstractions. We compare the most promising variants of RET and RCF heuristics to existing symbolic refinement verification of the Ecdar *tool. The results show that as the size of the system increases our heuristics are significantly less prone to exponential increase of time required by* Ecdar *to detect violations: in very large systems both "wide" and "narrow" violations are found up to 600 times faster and for extremely large systems when* Ecdar *timeouts, our heuristics are successful in finding violations.*

1 Introduction

Model-checking has been established as a useful technique for verifying properties of formal system models. The most notable obstacle in this field, *state-space explosion*, relates to the exponential growth of the state-space to be explored as the size of models increases. Over the last three decades a vast amount of research has attempted to combat this problem resulting in a plethora of techniques that reduce the number of states to be explored [1–3]. Various symbolic and reduction techniques (e.g. [4–8]) have become a ground for implementation of verification tools (CADP, NuSMV, Kronos, Spin, Uppaal, etc.), allowing them to handle a much larger domain of finite state and timed systems; however, for all cases symbolic and exhaustive verification still remains an expensive approach.

Counterexample detection techniques (e.g. [9]) can be used to even further avoid state-space explosion and facilitate a more efficient process of model verification. A prominent example in this area is the Counterexample-Guided Abstraction Refinement (CEGAR) [10] technique which has been intensely studied and applied to a variety of systems in model-checking including probabilistic systems [11], hybrid automata [12, 13], Petri net state equations [14] and timed automata [15–17]. The core idea is to automatically generate *abstraction models* (e.g. by reducing the amount of clocks), which may have a substantially smaller state-space, and verify them in a traditional model-checker to generate counterexamples if a property is not satisfied. The counterexamples are in turn used to refine the abstraction models.

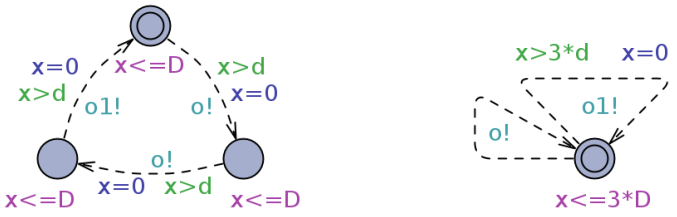
On the other hand, some counterexample detection techniques give up on the requirement of completeness and only explore part of the state-space, which no longer allows us to guarantee correctness but provides a powerful

mechanism for fault detection if one exists in the model. This is similar in approach to using the QuickCheck tool [18] for testing of Haskell program properties. Therefore, we believe a productive *development method* should consist of two steps: running multiple cheap and approximate counterexample detection algorithms early in the development for quick violation discovery and performing an expensive and exhaustive symbolic model-checking at the very end.

A very promising approach in counterexample detection methods is based on employing randomness. The first steps in that direction were made by [19, 20] where the state-space is explored by means of repeatedly performing *random walks*. With a sufficient amount of such walks an existing violation will eventually be found; nonetheless, designing efficient methods that excel at counterexample detection is not a trivial task. The difficulty lies in unintentional probabilities in the exploration methods that may lead to uneven coverage of the models' state-space. A recent example in the domain of *untimed systems* was done by [21] where the authors study verification of LTL properties and compare their random walk tactics, namely *continue walking* and *only accepting* and respective memory-efficient variants of those, to the tactics of [19].

A first attempt to use randomness in the setting of *timed systems* was made by [22], where a *Deep Random Search* (DRS) algorithm, which explores the state-space of a simulation graph of a *timed automaton* (TA) [23] in a symbolic manner, was presented. DRS performs an exhaustive exploration by means of random walks in a depth-first manner until a specified *cutoff* depth. Even though DRS conducts a complete search of the state-space, its computational advantage relies on detecting existing counterexamples quickly. In some sense DRS conforms to both steps of the above-mentioned *development method* - either counterexamples are detected potentially early in the search or, if none exist, the entire state-space is explored. DRS has been experimentally shown to outperform Open-Kronos and Uppaal model-checkers; however, the experiments do not compare DRS with Uppaal's *Random Depth First Search* (RDFS) - a powerful method for TA with strong fault detection potential.

In this paper we focus on carrying out random walks on networks of *timed I/O automata* (TIOA) for refinement checking as a quick and efficient falsification method. To improve performance we intend to work with the concrete semantics which relieves us from expensive computations of symbolic abstractions based on such data structures as *Difference Bounded Matrices* (DBM) [24]. The refinement verification helps to determine if a system specification can be successfully replaced by a single or even a number of other systems. Figure A.1 shows a common refinement application example where a detailed system (a) refines a more general specification of desired behavior (b). The detailed system models a token being passed around a ring. The



(a) Concrete system **(b)** Overall specification

Fig. A.1: Detailed automaton (a) refines overall specification (b).

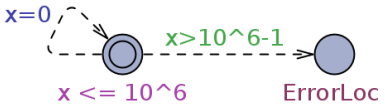


Fig. A.2: Timed I/O Automaton. Difficult case for SMC.

clock x ensures that the token is passed within the time bounds of `x>d` and `x <=D`. The overall specification requires the whole loop to be completed within `x<=3*D`.

An easy way to perform randomized exploration is to exploit the stochastic semantics of TIOA allowing the use of existing Statistical Model Checking (SMC) techniques [25, 26]. The idea of SMC is to produce a number of *sample traces* from a stochastic model, that are then statistically analyzed to estimate a probability that a random run of the model will satisfy a given property. Moreover, the estimate comes with a level of confidence which requires more sample traces for higher precision. The SMC method has been implemented in a number of tools, including Uppaal SMC [27] which uses stochastic timed automata (STA). For more details of this stochastic semantics see [27, 28]. Due to its simplicity, SMC is widely accepted in industrial applications where exhaustive model-checking is not feasible.

For the purpose of violation discovery however, SMC simulation techniques may not be a productive approach. Figure A.2 shows a trivial, yet very difficult case for SMC to detect if an `ErrorLoc` is ever reached. Due to the stochastic semantics SMC operates on, a delay is uniformly chosen between 0 and 10^6 making it nearly impossible to traverse "narrow guard" edges. The probability of reaching `ErrorLoc` in one step is $\frac{1}{2} * 10^{-6}$, thus it requires in average $2 * 10^6$ steps to reach that location. While such stochastic semantics allows for a model to mimic the behavior of a real system, counterexample detection methods require different heuristics in order to be efficient.

In this paper we present two lightweight, randomized and memory-less techniques for refinement checking of Timed I/O Automata: *Random Enabled Transition* (RET) and *Random Channel First* (RCF). Similarly to Uppaal SMC

and existing randomized techniques, our methods operate on concrete states and perform random walks through systems to detect violations. We show experimentally the potential of these algorithms on Milner's scheduler and Leader Election protocol with a varying number of nodes and compare their performance to those of existing symbolic and discrete state-space exploration methods - ECDAR and SMC for Timed I/O Automata. Our heuristics detect violations of the overall specification up to 600 times faster than ECDAR and scale better.

2 TIOA, Composition and Refinement

We now introduce key definitions of the formalism based on [29]. Let Clk be a finite set of clocks. A clock valuation over Clk is a mapping $u \in [Clk \mapsto \mathbb{R}_{\geq 0}]$. A guard is represented as a finite conjunction of expressions of the form $x \prec n$, where $x \in Clk$, $\prec$ is a relational operator $(<, \leq, >, \geq, =, \neq)$ and $n \in \mathbb{N}$. A set of such *guards* over Clk is denoted as $\mathcal{B}(Clk)$, whereas $\mathcal{P}(Clk)$ is used to denote a powerset of the clock set.

Definition 9 (Timed I/O Automaton)
A Timed I/O Automaton (TIOA) is a tuple $A = (Loc, q_0, Clk, E, Act, Inv)$ where Loc is a finite set of locations, $q_0 \in Loc$ is the initial location, Clk is a finite set of clocks that represent time, $E \subseteq Loc \times Act \times \mathcal{B}(Clk) \times \mathcal{P}(Clk) \times Loc$ is a set of edges, $Act = Act_i \oplus Act_o$ is a finite set of actions, partitioned into inputs and outputs respectively, and $Inv\colon Loc \to \mathcal{B}(Clk)$ is a set of location invariants.

An edge is a tuple $(q,\ a,\ \varphi,\ c,\ q') \in E$ where the source location is q, the action label is a, the constraint over clocks to be satisfied is φ, the clocks to be reset are c, and the target location is q'. The semantics of TIOA is given by a Timed I/O Transition System $S = (St, s_0, \Sigma, \rightarrow)$, where St is an infinite set of states, $s_0 \in St$ is the initial state, $\Sigma = \Sigma_i \oplus \Sigma_o$ is a finite set of actions and $\rightarrow\colon St \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times St$ is a transition relation (see [29] for complete definition).

An example of **Researcher** TIOA, shown in Figure A.3, contains three locations - `id0`, `id1` and `id2`. Input and output actions are denoted by `?` and `!` respectively. A **Researcher** can do some work `w!`(e.g. research) with at least **8** and at most **10** time units required to finish the job, defined as constraints on clock x: the guard `x>=8` on edge from `id0` to `id2` and invariant `x<=10` at location `id0`, respectively. Alternatively, if a researcher receives a

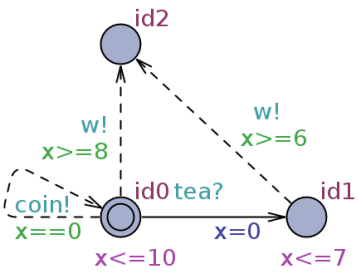


Fig. A.3: **Researcher** automaton.

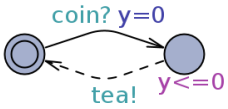


Fig. A.4: **`Machine`** specification.

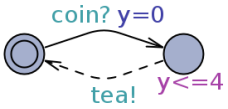


Fig. A.5: **`SlowMachine`** specification.

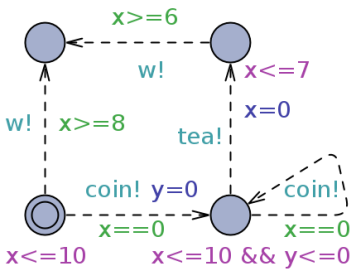
cup of tea (`tea?`) the work can be done faster - between **6** and **7** time units. However, for now this TIOA has a flaw in that the initial work progress, if any, is lost (due to the update `x=0`) when a researcher gets a cup of tea.

A run within TIOA is a sequence of *concrete states* defined as (l, u), where l is a location and u is a function that assigns values to all clocks. The following gives two sample runs ρ_1 and ρ_2 of the **`Researcher`**:

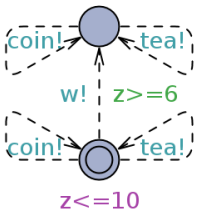
$$\rho_1 \equiv (\texttt{id0}, \texttt{x=0}) \xrightarrow{\texttt{9.27}} (\texttt{id0}, \texttt{x=9.27}) \xrightarrow{\texttt{w!}} (\texttt{id2}, \texttt{x=9.27})$$

$$\rho_2 \equiv (\texttt{id0}, \texttt{x=0}) \xrightarrow{\texttt{1.14}} (\texttt{id0}, \texttt{x=1.14}) \xrightarrow{\texttt{tea?}} (\texttt{id1}, \texttt{x=0}) \xrightarrow{\texttt{4.91}} (\texttt{id1}, \texttt{x=4.91}) \xrightarrow{\texttt{w!}} (\texttt{id2}, \texttt{x=4.91})$$

Parallel composition, a feature allowing to combine specifications, is an important aspect of refinement verification. An overall specification is often challenged to be refined by a number of parallelly composed systems. Consider a simplistic **`Machine`** component, shown in Figure A.4, which is responsible for providing tea immediately after the payment (`coin`) is received. It can be run in parallel (i.e. composed) with previously seen **`Researcher`** (Figure A.3) where both components are able to interact with each other and altogether act as a single system. To avoid state-space unfolding, composition is usually not constructed, but its behavior is deduced based on transition synchronization rules (see [29] for formal definition). For illustration purposes, the automaton which captures the overall behavior of parallelly composed **`Machine`** and **`Researcher`** components is given in Figure A.6 (a).



(a) **`Researcher || Machine`**.



(b) Overall specification **`Spec`**.

Fig. A.6: Composition (a) refines overall specification (b).

Note that only automata whose output action sets are disjoint may be

composed. Moreover, input and output edges that synchronize on identical signatures become output edges in a resulting composition (e.g. `coin` and `tea`). Such *internal synchronization* reflects both components advancing to new locations simultaneously. Since the composition component is now in control of when the tea is received, the work progress of a researcher can no longer be lost.

To capture the desired behavior of components a notion of *specification* is introduced. Its concept of *input-enabledness* reflects a belief that an input cannot be prevented from being sent to the system and thus requires an explicitly modelled behavior. To improve the modelling process, model-checking tools such as ECDAR treat unspecified behavior for inputs as location loops in the automaton.

Definition 10 (Specification)
A TIOTS $S = (St, s_0, \Sigma, \rightarrow)$ is a specification if each of its states $s \in St$ is input-enabled: $\forall i? \in \Sigma_i . \exists s' \in St.\ s \xrightarrow{i?} s'$.

The specification theory of TIOA supports a notion of *refinement* which if satisfied allows us to replace a specification with another one in every environment and obtain an equivalent system. For a specification S to refine specification T, both outputs and delays done by S must be matched by T, leading to a new pair of states in the refinement relation. Moreover, all inputs of T are required to be matched by S, which is always the case due to input-enabledness of specifications.

Definition 11 (Refinement)
A specification $S = (St^S, s_0, \Sigma, \rightarrow^S)$ refines a specification $T = (St^T, t_0, \Sigma, \rightarrow^T)$, written $S \leq T$, iff there exists a binary relation $R \subseteq St^S \times St^T$ containing $(s_0,\ t_0)$ such that for each pair of states $(s,\ t) \in R$ we have:

Input rule: whenever $t \xrightarrow{i?}{}^T t'$ for some $t' \in St^T$ then $s \xrightarrow{i?}{}^S s'$ and $(s',\ t') \in R$ for some $s' \in St^S$

Output rule: whenever $s \xrightarrow{o!}{}^S s'$ for some $s' \in St^S$ then $t \xrightarrow{o!}{}^T t'$ and $(s',\ t') \in R$ for some $t' \in St^T$

Delay rule: whenever $s \xrightarrow{d}{}^S s'$ for $d \in \mathbb{R}_{\geq 0}$ then $t \xrightarrow{d}{}^T t'$ and $(s',\ t') \in R$ for some $t' \in St^T$

Figure A.6 shows a refinement example where a **`Researcher || Machine`** composition (a) is challenged to refine a more general desired behavior specification **`Spec`** (b). Since the composition requires between **6** to **10** time units to perform the work, which is what the overall specification expects, the refinement relation holds. However, if the **`Researcher`** is composed with the **`SlowMachine`** from Figure A.5 instead, the tea is no longer provided immediately but requires up to **4** time units to be prepared. Performing the work

after getting the tea altogether now requires **11** time units at most. This is not allowed by the overall specification with invariant `z<=10` and thus refinement fails.

The refinement in the ECDAR tool is handled by using the UPPAAL TIGA engine [30] for verification of timed games. This engine searches for a winning strategy by playing a turn-based game between two players using the on-the-fly algorithm proposed in [31]. The first player, being the attacker, plays outputs of the left side and inputs of the right side of the refinement, while the second player, the defender, plays inputs of the left side and outputs of the right side. The refinement fails if the defender cannot match either a delay or an action performed by the attacker. The underlying data structure for the algorithm of [31] is based on *zones* which provides a *zone-based* symbolic abstraction, allowing to effectively store and manipulate states. Zones represent sets of clock valuations, defined as lower and upper bound constraints on clocks and on differences between each of the clocks. Unlike reachability analysis, refinement verification requires keeping track of a pair of states - one for each refinement side, which includes a single zone containing the union of clocks from both sides of the refinement relation. All newly discovered state pairs and already verified ones are stored in the *waiting* and *passed* data structures respectively, the latter of which allows us to guarantee termination and avoid repeated exploration of states.

3 Random Walk Heuristics

Conducting concrete-state based random walks means that we are no longer able to verify refinement but are rather looking for violations of one of the refinement rules. Verification of the delay rule is similar to the symbolic approach. Following the definition, it suffices to check if the refinement right side allows delaying at least as much as the left side. With a concrete state as a starting point it is easy to compute the *maximal delay* available for that state by selecting the smallest difference between the upper bound specified by the invariant and the current value for each individual clock. Since such computations are also necessary for determining transition's availability, for each encountered state pair we check if the maximal delay on the right side is at least as big as on the left side, thus potentially capturing more delay rule violations at a small cost.

To maintain quick state-space exploration, our random walks are completely memory free, i.e. no state pairs are stored in memory except for the current one. When a transition is taken, we advance to the target state pair and verify either input or output rule based on the action type of the transition. Due to input-enabledness, an input transition may only result in the discovery of a new state pair, whereas an output transition on the left side, if

not followed by the right side, can provide a counterexample. Moreover, not storing any information about already visited states introduces two issues: termination guarantee and repeated exploration of the states.

Termination In the setting of concrete-state random walks, revisiting already explored state pairs is not necessarily a bad thing; in fact, it can be beneficial as it may lead to traversal of other, yet unseen, transitions. Termination on the other hand requires certain conditions. Upon reaching a state with either no outgoing transitions or no eventually enabled (after performing a delay) transitions we terminate the random walk and issue a new one. This, however, becomes a problem for cyclic systems where above-mentioned conditions may never occur, resulting in an infinite exploration. We approach the termination problem in a straightforward way by supplying random walks with a parameter of *steps* (number of transitions) that can be taken before a walk is terminated. Ideally, this parameter should be dynamically adapted to the target system; however, finding the optimal value is far beyond from trivial (e.g. see [32]). Therefore, we limit ourselves to a predefined (static) number of steps.

3.1 Selecting transition

During a random walk through the model, the actions of both delaying and traversing transition are made in sequence. We, however, reverse the process such that the concrete delay is selected after the target transition is chosen. As a result, not only do delays not determine transition choice, but a delay is no longer made if there are no transitions available. Given that the delay rule is checked by comparison of *maximal delays* of refinement sides, this strategy (of choosing transition first) makes sense as with no available transitions no other refinement rules can be violated. We propose two heuristics for selecting transitions.

The idea of the *Random Enabled Transition* (RET) heuristic is to first compute all eventually enabled transitions, i.e. transitions which are either currently available or will become such after a delay, for a given state of the refinement left side. Contrary to the refinement input rule, we consider input transitions starting from the left side as due to input-enabledness they can never violate refinement relation, but can only lead to new, potentially unexplored, state pairs. Next, we uniformly choose one of the computed transitions as a target for traversal. The counterexample is found when the right side cannot match an output transition.

Profiling has shown that computing eventually enabled transitions is the most resource demanding operation in our random walks. It needs to consider all parallelly composed automata and construct transitions on the fly. Given a concrete state it is necessary to check potential availability, i.e. if guards are satisfied, for each edge by computing lower and upper bounds

that correspond to minimal and maximal delays after which a transition is enabled. To reduce the total amount of such computations we propose an alternative heuristic for choosing transitions - *Random Channel First* (RCF). It chooses a random channel (same as action) from the list of all channels and computes enabled transitions only for that channel. If none exist, the selected channel is removed from the list. The process is repeated until either transitions are found or the list of channels is exhausted, where the latter option leads to *termination* of the random walk. If transitions are discovered, a random one is uniformly chosen for traversal.

3.2 Selecting delay

Next, we need to select a concrete delay, i.e. value to increase all clocks by, before traversing a transition as it potentially affects further choices. With target transition being selected first, the choice of delay is made within availability bounds of the transition which are computed during transition selection. This keeps the process lightweight as no additional computations are required. Choosing a target transition prior to delaying also allows for excluding the width (size) of the edge's guard from affecting the probability for that edge to be explored. A delay for the automaton from Figure A.2 would therefore depend on a chosen transition and be in either of the two ranges - $[0; 10^6]$ or $(10^6 - 1; 10^6]$. In comparison to SMC our heuristics have a probability of $\frac{1}{2}$ to traverse the edge leading to `ErrorLoc` thus requiring 2 runs in average to discover the error. This leads to a better state-space coverage and increases the chance to detect counterexamples since "narrow" and "wide" edges become equally easy to traverse.

Initially, we selected the delay uniformly from the transition's range of available delays. We believe however that selecting lower bound (LB) or upper bound (UB) is often more efficient for violation detection. This is because the prevailing amount of practical model-checking applications is concerned with either meeting deadlines, i.e. something that has an upper limit, or satisfying minimal requirements, i.e. something that cannot be done faster than specified. For example, the overall specification from Figure A.6 (b) ensures both upper and lower time limits to be followed by a more concrete system. Similarly in QuickCheck [18] the random selection of datatype values is biased towards base-elements (empty list, empty tree, etc.) because they are more likely to be the source of errors. Thus, we expect a more corner-case oriented delay choice distribution (e.g. 40% LB, 20% uniform, 40% UB) to show better results at violation detection.

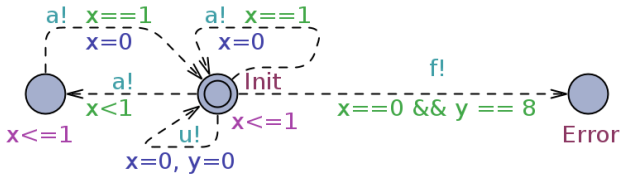


Fig. A.7: Difficult case for RCF.

3.3 RET vs RCF

Since the RCF heuristic partitions the computation of eventually enabled transitions into smaller chunks, which are based on channel, and chooses a target transition as soon as one of these chunks yields a result, it is less computationally demanding than RET. For models with outdegree of at least two edges with different channels, RCF in average will perform fewer expensive operations to compute transitions which implies a faster exploration of the state-space; however, due to underlying probabilities this is not always the case.

Consider the automaton from Figure A.7 where the `Error` location represents a counterexample. At the initial location `Init` both the values of clocks `x` and `y` are set to 0 and the output edge with action `f` is not available until `x==0` and `y==8`. Moreover, the invariant (`x<=1`) on `Init` restricts us from directly delaying until the `f` edge becomes available. This leaves three enabled edges: two of action `a`, both of which increase the clock `y` by 1 unit upon returning to `Init`, and one of action `u`, which is "undesired" in a sense that it prohibits the walk from reaching `Error` by resetting clock `y` and must therefore be avoided during exploration.

While clock `y` is less than 8, only channels `a` and `u` can yield a result for a target transition. The probabilities for RCF and RET to choose edges for these channels is shown in Table A.1. RCF heuristic randomly chooses one of two channels at a 50% probability, leaving a 50% chance to traverse either one of the `a!` edges or the only existing edge for channel `u`, which "resets" the model back to its initial state. On the other hand, choosing randomly amongst all eventually enabled transitions regardless of channel, RET selects any of the three edges at a probability of 33%. To reach the `Error` location either of the two `a!` edges must be taken 8 times in a row, followed by the `f!` edge. The probability of doing so in one *attempt* for RET is $0.67^8 * 0.25 \approx \frac{102}{10000}$, whereas for RCF it is $0.5^8 * 0.33 \approx \frac{13}{10000}$. After traversing the "undesired" edge, a random walk continues making new *attempts* until either the vi-

Table A.1: RET and RCF probabilities to traverse edges while `y<8`.

Action	RET	RCF
`a!` (`x<1`)	33.3%	50%
`a!` (`x==1`)	33.3%	
`u!`	33.3%	50%

olation is found or the number of allowed steps is made. Given the probabilities, RCF requires 769 attempts in average to reach `Error` compared to an average of 98 attempts for RET.

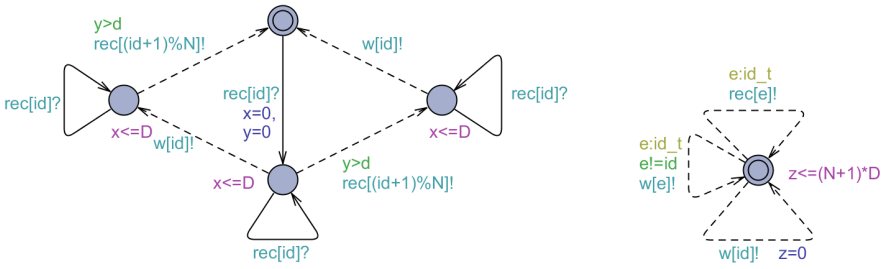
3.4 Delay probability distribution changes

The drawback of the static delay choice proposed in Section 3.2 is that such (or any) static distribution (40% LB, 20% uniform, 40% UB) naturally favors some models more than others in terms of error detection. In fact, some sophisticated systems might benefit the most from delaying only LB or UB; however, it might be impossible to derive this knowledge from a static analysis of the system.

Algorithm 1 Check refinement

```
1: function checkRefinement
2:     chanceUB ← 0.5, chanceLB ← 0.5
3:     while violation not found do
4:         perform random walk with chanceLB and chanceUB
5:         if violation found then return false
6:         else
7:             chanceUB += 0.1;                    ▷ Increase UB by 10%
8:             if (chanceUB > 1) then chanceUB = 0;
9:             chanceLB = 1 − chanceUB;
```

To fight this, we propose a strategy where each random walk has a different delay choice distribution, as shown in Algorithm 1. First, a random walk is executed where all the delays follow 50% LB / 50% UB distribution. If a violation is not found, a new random walk is issued where the probability to delay LB is decreased and probability to delay UB is increased by 10%, resulting in 40% LB / 60% UB. Upon reaching a probability distribution which guarantees the choice of an upper bound value (0% LB / 100% UB), the next random walk has its probabilities "flipped", s.t. only the lower bound value is chosen for the delay. The process continues until the violation is found. Naturally, if a random walk with the most efficient probability distribution for a target model is unsuccessful at finding a violation, it will take another 11 random walks to reach that probability distribution again. However, the main drawback of this strategy is its inability to detect the "in between" violations as only bounds of the potential delay range are considered; nonetheless, while always missing a particular kind of violation, we believe this technique will often be substantially more efficient than others.



(a) Node template.

(b) Overall specification template.

Fig. A.8: Real-time version of Milner's Scheduler. Templates for the ECDAR tool.

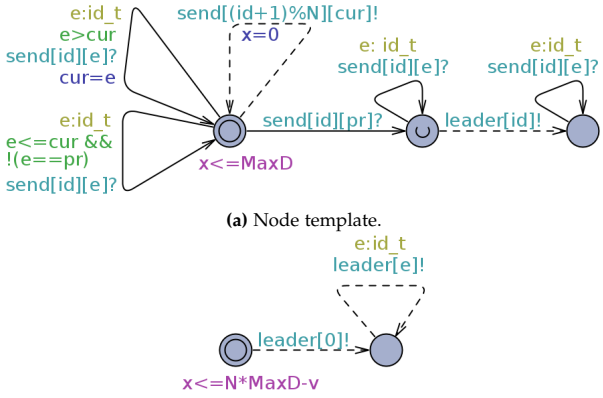
4 Test setting

The experiments are performed on the models of Milner's scheduler [33] and Leader Election protocol [34], which operate on a ring topology and can be instantiated for an arbitrary number of nodes that communicate in sequence.

4.1 Milner's scheduler

We analyze a real-time version of Milner's scheduler [33], where each node N_i can perform two actions in parallel: do some work by outputting on $\mathbf{w_i!}$ and pass the token to the next node N_{i+1} in the sequence by outputting on $\mathbf{rec_{i+1}!}$. Figure A.8 shows a node template (a) and a template for the overall specification (b) that a ring of nodes has to refine. Templates allow multiple instances of the same model as also used in UPPAAL [35]. Each node starts at a location where it waits to receive a token before any actions can be taken. As soon as the token is received clocks are reset and all further actions are limited by a lower bound of `d` and an upper bound `D` represented by guards and invariants respectively.

Note that the first node of the system has to be instantiated with a different initial location (bottom one) to represent the initial ownership of the token. The overall specification on the other hand only ensures that `w0!`, i.e. work done by initial node, requires at most `(N+1)*D` time units. Later in our experiments, we modify the overall specification such that it is violated in order to create counterexamples that can be detected by RET, RCF and SMC. To do so, we modify the invariant of the overall specification to be `z<=(N+1)*D*(1-v)`, s.t. $\{$`v` $\in \mathbb{R}: 0 \leq$ `v` $\leq 1\}$ where `v` is the desired violation size in percentage. The higher the value of `v` the wider, and therefore easier to detect, violation is created. Apart from different node amounts, we also manipulate the lower bound variable on guards (`d`), the smaller values of which drastically increase the state-space.



(a) Node template.

(b) Overall specification template.

Fig. A.9: Leader Election protocol templates for the ECDAR tool.

4.2 Leader Election protocol

The Leader Election protocol has each of its nodes assigned a unique *priority* in addition to *id*. The goal of the protocol is to elect a leader with the highest *priority* by having nodes pass their current *priority* to the next node in sequence. If the *priority* received by a node is higher than its own, the node records that *priority* and further only sends it to the next node instead of its own *priority*. Otherwise, the received *priority* is discarded. Upon receiving its own *priority* the node knows it can claim the leadership since that *priority* has travelled one full round without being discarded and is thus the highest.

The templates for this protocol are shown in Figure A.9. The overall specification (b) ensures that only the correct node (a) can declare itself a leader, and only within `N*MaxD-v` time units, s.t. $\{\texttt{v} \in \mathbb{Z} : 0 \leq \texttt{v} \leq \texttt{N*MaxD}\}$, where `v` is used to modify specification to introduce refinement violations. Here, two-dimensional channel arrays, e.g. `send[id][pr]`, are used as way of value passing, where the first and the second indices represent node *id* and *priority* respectively. The `cur` variable, representing the highest received priority, is initialized to `pr` (own priority) for each node. Contrary to Milner's scheduler, this protocol does not constrain nodes to acting only after having received the token; instead, any node is free to send its priority at all times.

4.3 Implementation

We have implemented a Java prototype of both the RET and RCF heuristics for refinement checking of TIOA. For a more fair comparison of our heuristics with SMC, we reimplemented SMC in Java for the network of TIOA with the stochastic semantics. Table A.2 gives an overview on performance differences

Table A.2: Average time (in seconds) to detect violation for Milner's scheduler.

Settings	Java SMC	Uppaal SMC
N=8, d=20, v=6%	1.720	3.413
N=12, d=20, v=6%	33.869	57.762

Table A.3: Each cell represents an average time (in sec) to discover a violation calculated over all discovered violations within 60 minutes in Milner's scheduler. Not found (nf) cells represent no discovered violations.

Settings		RET-U	RET-PD	RET	RCF-U	RCF-PD	RCF	SMC
N=8 d=20	*v=2%*	nf	0.042	0.011	nf	0.024	**0.007**	nf
	v=4%	21.577	0.008	0.003	12.590	0.005	**0.002**	50.832
	v=6%	0.558	0.004	0.003	0.361	0.003	**0.002**	1.720
N=8 d=4	*v=2%*	nf	0.078	0.010	nf	0.043	**0.005**	nf
	v=4%	491.800	0.044	0.010	1506.872	0.026	**0.005**	891.425
	v=6%	*58.688*	0.033	0.010	42.996	0.017	**0.005**	96.811
N=12 d=20	*v=2%*	nf	0.655	0.030	nf	0.336	**0.017**	nf
	v=4%	2770.389	0.082	0.017	1376.237	0.043	**0.010**	2882.110
	v=6%	26.082	0.021	0.008	13.564	0.012	**0.005**	33.869
N=12 d=4	*v=2%*	nf	2.056	0.032	nf	0.886	**0.017**	nf
	v=4%	nf	0.851	0.031	nf	0.440	**0.017**	nf
	v=6%	nf	0.501	0.031	nf	0.254	**0.017**	nf

between Java and that of Uppaal C++ implementation of SMC. Surprisingly, Java SMC appeared to be faster, however this is most likely due to it being a prototype which does not retain all the features of Uppaal SMC. For the rest of the paper we will be using Java SMC as it is not substantially different.

To use SMC in a refinement setting, we transform refinement into a reachability problem by constructing a *complement* automaton of the refinement right side and composing it with the left side.

5 Experiments

To understand how delay choice influences violation detection, we compare the performance of three variants of each heuristic and the SMC approach. The results are reported in Table A.3 for Milner's scheduler where each case ran for 60 minutes. RET-U and RCF-U did a *uniform* delay choice, RET-PD and RCF-PD delayed based on a *predefined distribution* of 40% LB, 20% uniform, 40% UB, and RET and RCF had changing probability distributions, but could miss violations requiring "intermediate delays", as described in Section 3.4.

It is clear that delay choice strategies have a large impact on the efficiency of random walks. Both the SMC approach and our heuristics with uniform delay choice (RET-U, RCF-U) have the weakest potential in terms of coun-

Table A.4: Each cell represents an average time (in sec) to discover a violation calculated over all discovered violations within 60 minutes in Milner's Scheduler. The $v = 0\%$ case can only be verified by the complete exploration of ECDAR.

Settings		ECDAR	RET	RCF
N=50 *d=20*	*v=0%*	**0.619**	-	-
	v=2%	0.686	0.638	**0.314**
	v=4%	0.688	0.487	**0.249**
	v=6%	0.689	0.360	**0.176**
N=50 *d=10*	*v=0%*	**1.576**	-	-
	v=2%	2.252	0.692	**0.326**
	v=4%	2.208	0.613	**0.291**
	v=6%	2.182	0.547	**0.255**
N=50 *d=4*	*v=0%*	**160.015**	-	-
	v=2%	224.724	0.688	**0.322**
	v=4%	274.632	0.621	**0.292**
	v=6%	295.818	0.576	**0.268**

Settings		ECDAR	RET	RCF
N=100 *d=20*	*v=0%*	**3.622**	-	-
	v=2%	4.050	2.791	**1.304**
	v=4%	3.942	2.206	**1.024**
	v=6%	3.974	1.701	**0.776**
N=100 *d=10*	*v=0%*	**9.510**	-	-
	v=2%	13.367	2.873	**1.302**
	v=4%	13.252	2.686	**1.194**
	v=6%	12.832	2.383	**1.080**
N=100 *d=4*	*v=0%*	**2631.751**	-	-
	v=2%	693.688	2.856	**1.279**
	v=4%	695.181	2.721	**1.231**
	v=6%	689.754	2.490	**1.102**

terexample detection and are strongly affected by the size of the violation. While "wide" violations are found relatively quickly, "narrow" counterexamples (*v*=2%) were not discovered at all. Therefore, the low efficiency of SMC, RET-U and RCF-U approaches makes their practical application not feasible for a number of nodes higher than 12. On the other hand, RET-PD, RET, RCF-PD and RCF are significantly quicker at discovering violations and less sensitive to increasing the number of nodes or decreasing the d variable, both of which explode the state-space. The delay choice based on the predefined distribution (RET-PD and RCF-PD) was, as expected, superior to uniform choice and enabled detection of even "narrow" violations. The most efficient appears to be RET and RCF heuristic variants with changing probabilities, which have also shown the smallest difference in time for detection of "wide" and "narrow" violations.

We further compare the most promising RET and RCF heuristics with ECDAR on a large number of nodes and report results in Table A.4. Increasing the number of nodes or especially decreasing d significantly increases time needed by ECDAR for verification. Contrary to that, RET and RCF are not so sensitive to the change of d which shows that due to probability changes our heuristics perform almost equally well on "narrow" and "wide" edge systems. For $N = 100$ and $d = 4$ ECDAR takes more than 10 minutes to detect the violation, whereas RET and RCF require just under 3 and 1.3 seconds respectively. Surprisingly, complete symbolic refinement verification in ECDAR in case of $v = 0\%$ is still feasible on such high number of nodes as 50 and 100. Thus, the use of our proposed development method is supported: first RET and RCF can be used to quickly detect possible violations, and once no further violations are found using our heuristics an expensive and complete verification by ECDAR is to be conducted.

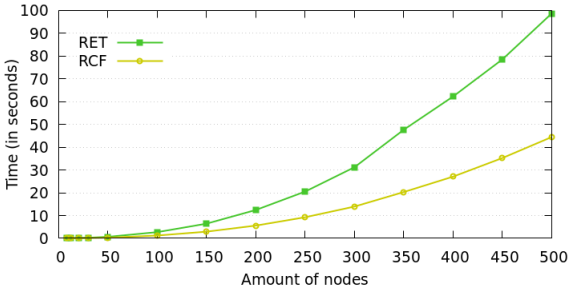


Fig. A.10: RET and RCF comparison on Milner's scheduler with $d = 4$ and $v = 2\%$

Table A.5: Each cell represents an avg. time (in sec) to discover a violation calculated over all discovered violations within 60 minutes in Leader Election protocol. The $v = 0$ case can only be verified by the complete exploration of ECDAR. Not found (nf) cells represent no discovered violations.

Settings		ECDAR	RET	RCF
N=5	*v=0*	**0.103**	-	-
	v=2	**0.127**	1.411	0.403
	v=4	0.130	0.080	**0.024**
	v=6	0.085	0.009	**0.003**
N=7	*v=0*	nf	-	-
	v=2	nf	102.653	**26.617**
	v=4	nf	4.140	**0.722**
	v=6	nf	0.345	**0.073**

Settings		ECDAR	RET	RCF
N=6	*v=0*	**17.190**	-	-
	v=2	18.170	11.392	**2.916**
	v=4	15.952	0.576	**0.138**
	v=6	8.695	0.059	**0.015**
N=8	*v=0*	nf	-	-
	v=2	nf	**170.782**	172.880
	v=4	nf	38.217	**5.166**
	v=6	nf	2.113	**0.340**

In Figure A.10 the performance of RET and RCF for Milner's scheduler increasing number of nodes is compared in the difficult setting of $d = 4$ and $v = 2\%$ which significantly reduces chances to detect violations. The results are very encouraging: even for 500 nodes RET and RCF manage to discover violations in an average of under 100 and 50 seconds respectively.

We now compare most promising variants of RET and RCF (with changing probabilities) against ECDAR on a much heavier, non tokenized Leader Evaluation protocol. The results (shown in Table A.5) demonstrate a severe state-space explosion: even for 7 nodes ECDAR is not able to conclude verification within an hour. On the positive note, RET and RCF are able to handle up to 10 nodes; however, in comparison to Milner's scheduler, here the "width" of the violation has a much stronger impact on the performance. Moreover, the exponential growth of channels (`send[id][e]`) makes the RCF heuristic much more favorable.

To further examine the efficiency of our heuristics to quickly detect violations during iterative development, we perform mutation testing on Leader Election protocol. Table A.6 reports the results where either one ($M^{\exists}_{1-4}$) or all ($M^{\forall}_{1-4}$) nodes have been replaced with a certain type of mutant, s.t. the

Table A.6: Mutation testing for Leader Election protocol. Each cell represents an avg. time (in sec) to discover a violation calculated over all discovered violations within 60 minutes. Not found (nf) cells represent no discovered violations.

Settings		ECDAR	RET	RCF
N=6	$M_1^{\exists}$	38.204	51.704	**11.697**
	$M_2^{\exists}$	25.183	0.002	**0.001**
	$M_3^{\exists}$	19.709	0.002	**0.001**
	$M_4^{\exists}$	18.007	0.002	**0.001**
N=6	$M_1^{\forall}$	12.592	0.077	**0.016**
	$M_2^{\forall}$	11.452	0.006	**0.001**
	$M_3^{\forall}$	10.643	0.003	**0.001**
	$M_4^{\forall}$	11.183	0.005	**0.001**

Settings		ECDAR	RET	RCF
N=7	$M_1^{\exists}$	nf	563.254	**125.991**
	$M_2^{\exists}$	nf	0.005	**0.001**
	$M_3^{\exists}$	nf	0.005	**0.001**
	$M_4^{\exists}$	687.573	0.005	**0.001**
N=7	$M_1^{\forall}$	nf	0.054	**0.011**
	$M_2^{\forall}$	nf	0.007	**0.001**
	$M_3^{\forall}$	nf	0.006	**0.001**
	$M_4^{\forall}$	35.230	0.001	**0.001**

refinement relation is violated. We have tried a mutant with the initial location's invariant bound doubled (M_1), a mutant that always sends its own *priority* instead of the recorded one (M_2), a mutant that forgets to record the received *priority* (M_3) and a mutant that records its own *id* instead of the received *priority* (M_4).

The time to discover violation with mutants M_2-M_4 is surprisingly small, which persists for even higher amount of nodes with very small increments in time. This occurs due to the modifications in these mutants in different ways leading to an overall inability of the "node ring" to elect a leader, which most of the time can be detected with only a single random walk. Mutant M_1 on the other hand does not prevent leadership from being correctly declared, but creates the possibility of it happening too late, i.e. violating the time requirement imposed by the overall specification. In cases where only one such mutant is present in the "ring" ($M_1^{\exists}$) it is significantly harder to detect violation for both ECDAR, due to the state-space growth, and our heuristics, due to decreasing underlying probabilities to find a violation.

Overall, for both of the models RCF appears to be noticeably faster than RET. This is caused by frequently occurring states with the outdegree of at least 2 transitions for different channels, which helps RCF to avoid a lot of expensive transition computations. This difference is especially large for Leader Election protocol, where the amount of channels grows exponentially to the amount of nodes. The general tendency is such that our heuristics are much less affected by state-space explosion than symbolic verification using ECDAR.

The complete model, test results and Java prototype code are available at `http://www.cs.aau.dk/~ulrik/submissions/982983/SETTA2020.zip`.

6 Conclusions and Future Work

We have presented what we believe to be the first randomized technique for refinement checking of Timed I/O Automata by means of random walks. Our two heuristics RET and RCF provide a fast and scalable way of detecting counterexamples, the benefits of which are most noticeable in large systems where the memory demands of symbolic verification are high. Such techniques are best used for quick falsification to save time during development of large and industrial sized systems. If no errors are found, a long and expensive complete symbolic verification can be conducted.

The experiments have shown that the choice of delays can strongly influence the efficiency of the technique. The most efficient and scalable variations of RET and RCF heuristics appeared to be the ones based on the adaptive approach, s.t. the delay choice distribution changes based on the outcome of the previous run. We anticipate that some models may even require the delay choice heuristic to be different for each state while for other systems it might suffice delaying according to the same distribution. Therefore, we believe that as more techniques appear, a successful violation detection strategy will be to run multiple heuristics in parallel (see e.g [36]).

The direction for the future work is to test RET and RCF on more models to see if these heuristics are efficient or different strategies are required. Our methods can also be applied for real time model-checking of other analysis problems than refinement. Furthermore, a better performance of the heuristics can potentially be achieved by supplying random walks with the dynamic number of steps based on a static analysis of the model and/or certain heuristics that manipulate the depth of each walk based on the outcome of the previous one.

References

[1] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "Partial-Order Reduction in Symbolic State Space Exploration," in *Computer Aided Verification*, O. Grumberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 340–351.

[2] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek, "Lower and Upper Bounds in Zone Based Abstractions of Timed Automata," in *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 312–326.

[3] C. Norris IP and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1, pp. 41–75, 1996.

[4] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi, "Efficient Timed Reachability Analysis Using Clock Difference Diagrams," in *Computer Aided Verification*, N. Halbwachs and D. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 341–353.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 1020 States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[6] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction," in *Proceedings Real-Time Systems Symposium*, 1997, pp. 14–24.

[7] J. Lind-Nielsen, H. R. Andersen, G. Behrmann, H. Hulgaard, K. Kristoifersen, and K. G. Larsen, "Verification of Large State/Event Systems Using Compositionality and Dependency Analysis," in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Steffen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 201–216.

[8] A. Valmari, "A Stubborn Attack on State Explosion," in *Computer-Aided Verification*, E. M. Clarke and R. P. Kurshan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 156–165.

[9] S. Kupferschmid, M. Wehrle, B. Nebel, and A. Podelski, "Faster Than Uppaal?" in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 552–555.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," in *Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169.

[11] H. Hermanns, B. Wachter, and L. Zhang, "Probabilistic CEGAR," in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 162–175.

[12] A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski, "Multiple shooting, cegar-based falsification for hybrid systems," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14. New York, NY, USA: Association for Computing Machinery, 2014.

[13] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based cegar for rectangular hybrid systems," *Formal Methods in System Design*, vol. 46, no. 2, pp. 105–134, 2015.

[14] H. Wimmel and K. Wolf, "Applying cegar to the petri net state equation," in *Tools and Algorithms for the Construction and Analysis of Systems*,

P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 224–238.

[15] T. Nagaoka, K. Okano, and S. Kusumoto, "An abstraction refinement technique for timed automata based on counterexample-guided abstraction refinement loop," *IEICE TRANSACTIONS on Information and Systems*, vol. 93, no. 5, pp. 994–1005, 2010.

[16] F. He, H. Zhu, W. N. Hung, X. Song, and M. Gu, "Compositional abstraction refinement for timed systems," in *2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 2010, pp. 168–176.

[17] K. Okano, B. Bordbar, and T. Nagaoka, "Clock number reduction abstraction on cegar loop approach to timed automaton," in *2011 Second International Conference on Networking and Computing*. IEEE, 2011, pp. 235–241.

[18] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 46, no. 4, p. 53–64, May 2011.

[19] R. Grosu and S. A. Smolka, "Monte Carlo Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Halbwachs and L. D. Zuck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 271–286.

[20] J. Oudinet, A. Denise, M.-C. Gaudel, R. Lassaigne, and S. Peyronnet, "Uniform Monte-Carlo Model Checking," in *Fundamental Approaches to Software Engineering*, D. Giannakopoulou and F. Orejas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 127–140.

[21] K. Larsen, D. Peled, and S. Sedwards, "Memory-Efficient Tactics for Randomized LTL Model Checking," in *Verified Software. Theories, Tools, and Experiments*, A. Paskevich and T. Wies, Eds. Cham: Springer International Publishing, 2017, pp. 152–169.

[22] R. Grosu, X. Huang, S. A. Smolka, W. Tan, and S. Tripakis, "Deep Random Search for Efficient Model Checking of Timed Automata," in *Composition of Embedded Systems. Scientific and Industrial Issues*, F. Kordon and O. Sokolsky, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 111–124.

[23] R. Alur and D. Dill, "The theory of timed automata," in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Springer, 1992, pp. 45–73.

[24] D. L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," in *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 197–212.

[25] K. Sen, M. Viswanathan, and G. Agha, "Statistical Model Checking of Black-Box Probabilistic Systems," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 202–215.

[26] H. L. S. Younes and R. G. Simmons, "Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling," in *Computer Aided Verification*, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 223–235.

[27] 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, pp. 397–415, 2015.

[28] N. Bertrand, P. Bouyer, T. Brihaye, and P. Carlier, "When are stochastic transition systems tameable?" *Journal of Logical and Algebraic Methods in Programming*, vol. 99, pp. 41 – 96, 2018.

[29] 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: Association for Computing Machinery, 2010, p. 91–100.

[30] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Time for Playing Games!" in *Computer Aided Verification*, W. Damm and H. Hermanns, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 121–125.

[31] 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*, M. Abadi and L. de Alfaro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 66–80.

[32] G. Behrmann, K. G. Larsen, and R. Pelánek, "To Store or Not to Store," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 433–445.

[33] R. Milner, *A Calculus of Communicating Systems*. Berlin, Heidelberg: Springer-Verlag, 1982.

[34] A. David, K. G. Larsen, A. Legay, M. H. Møller, U. Nyman, A. P. Ravn, A. Skou, and A. Wasowski, "Compositional Verification of Real-Time Systems Using Ecdar," *International Journal on Software Tools for Technology Transfer*, vol. 14, pp. 703–720, 2012.

[35] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer, 2004, pp. 200–236.

[36] J. I. Rasmussen, G. Behrmann, and K. G. Larsen, "Complexity in Simplicity: Flexible Agent-Based State Space Exploration," in *Tools and Algorithms for the Construction and Analysis of Systems*, O. Grumberg and M. Huth, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 231–245.

Paper B

Randomized Reachability Analysis in UPPAAL: Fast Error Detection in Timed Systems

Andrej Kiviriga, Kim Guldstrand Larsen and Ulrik Nyman

The paper has been published in the
International Journal on Software Tools for Technology Transfer 2022.

The layout has been revised.

Abstract

Randomized reachability analysis is an efficient method for detection of safety violations. Due to the under-approximate nature of the method, it excels at quick falsification of models and can greatly improve the model-based development process: using lightweight randomized methods early in the development for the discovery of bugs, followed by expensive symbolic verification only at the very end. We show the scalability of our method on a number of Timed Automata and Stopwatch Automata models of varying sizes and origin. Among them, we revisit the schedulability problem from the Herschel-Planck industrial case study, where our new method finds the deadline violation three orders of magnitude faster: some cases could previously be analyzed by statistical model checking (SMC) in 23 hours and can now be checked in 23 seconds. Moreover, a deadline violation is discovered in a number of cases that where previously intractable. We have implemented the Randomized reachability analysis – and made it available – in the tool Uppaal. *Finally we provide an evaluation of the strengths and weaknesses of Random reachability analysis exploring exactly which types of model features hamper method's efficiency.*

1 Introduction

The problem of state space explosion is the major issue keeping formal verification of industrial sized models from becoming a truly impactful technology. This paper presents a method, Random Reachability Analysis, which can help to combat state-space explosion in one particular way. The method searches the state space using randomization, the effect of this is a method that is very efficient at finding bug in most systems, but cannot prove the safety of a system. Throughout the process of developing formal models, an array of sanity queries can be used in the same way as unit tests in software development. Verifying these queries repeatedly between each addition to the model can be prohibitively time consuming, especially for complex systems that often grow large and become difficult to analyze. The method presented in this paper is exactly a solution to this problem.

The main contribution of this paper is the implementation of randomized reachability analysis in the tool Uppaal. Randomized reachability analysis is a non-exhaustive efficient technique for the detection of errors (safety violations). The work was inspired by [1] where similar randomized analysis was applied to refinement checking. The method can analyse Timed Automata and Stopwatch Automata models with the features already supported by Uppaal. The randomized approach is based on repeated exploration of the model by means of *random walks* and was inspired by [2]. It explores the state space in a light and under-approximate manner; hence, it can only perform conclusive verification when a single trace can demonstrate a prop-

erty. However, our randomized method excels at reachability checking and in many cases outperforms existing model-checking techniques by up to several orders of magnitude. The benefits are especially notable in large systems where traditional model-checking is often intractable due to the state space explosion problem.

Randomized reachability analysis is particularly useful for an *efficient development process*: running cheap, randomized methods early in the development to discover violations and performing an expensive and exhaustive verification at the very end. Randomized reachability analysis supports the search for *shorter* traces which improves the usability of discovered traces in debugging the model. We have implemented randomized reachability analysis – and made it available – in the tool UPPAAL[1] [3]. Unfortunately, our randomized methods are not a panacea and there are certain types of model features that the method is not well suited for exploring. These are discussed in Section 11 on strengths and limitations, divided into three categories. All categories relate to some way in which a certain part of the state space is potentially hard for the method to explore.

Timed Automata models can also be used in the domain of schedulability, which deals with resource management of multiple applications ranging from warehouse automation to advanced flight control systems. Viewing these systems as a collection of tasks, schedulability analysis allows us to optimize usage of resources, such as processor load, and to ensure that tasks finish before their deadline. A traditional approach in preemptive priority-based scheduling is that of the worst-case response time (WCRT) analysis [4, 5]. It involves estimating worst case scenarios for both the execution time of a task and the blocking time a task may have to spend waiting for a shared resource. Apart from certain applicability limitations, classical response time analysis is known to be over-approximate which may lead to pessimistic conclusions in that a task may miss its deadline, even if in practice such a scenario could be unrealizable. Model-based approach is a prominent alternative for verification of schedulability [6–9] as it considers such parameters as offsets, release times, exact scheduling policies, etc. Due to this, the model-based approach is able to provide a more exact schedulability analysis.

We continue the effort in using a model-based approach and the model checker UPPAAL to perform a Stopwatch Automata based schedulability analysis of systems [10]. Specifically, we re-revisit the industrial case study of the ESA Herschel-Planck satellite system [9, 11]. The Danish company Terma A/S [12] developed the control software and performed the WCRT analysis for the system. The case we analyse consists of 32 individual tasks being executed on a single processor with the policy of fixed priority preemptive scheduling. In addition, a combination of priority ceiling and priority in-

[1] `https://uppaal.org/downloads/`

Table B.1: Summary of schedulability of the Herschel-Planck system.

$f = \frac{\text{BCET}}{\text{WCET}}$	0-71%	72-80%	81-86%	87-90%	90-100%
Symbolic MC:	maybe	maybe	maybe	n/a	**Safe**
Statistical MC:	**Unsafe**	maybe	maybe	maybe	maybe
Randomized MC:	**Unsafe**	**Unsafe**	maybe	maybe	maybe

heritance protocols is used, which in essence makes the priorities dynamic. Preemptive scheduling is encoded in the model with the help of stopwatches which allow to track the progress of each task and stop it when the task is preempted. In UPPAAL, existing symbolic reachability analysis for models with stopwatches is over-approximate [13], which may provide spurious traces. In such models, our randomized reachability analysis allows us to obtain exact, non-spurious traces to target states.

In the previous work of [9] the schedulability of Herschel-Planck was "successfully" concluded, but with an unrealistic assumption of each task having a fixed execution time (ET). To improve on this, the analysis of [11] was carried out with each of the tasks given a non-deterministic execution time in the interval of Best Case and Worst Case ETs $[\text{WCET}, \text{BCET}]$. Unfortunately, interval based execution times, preemption and shared resources that impose dependencies between tasks, makes schedulability of systems like Herschel-Planck undecidable [14].

Even in the presence of unschedulability, two model-checking (MC) techniques were used in [11] to either verify or disprove schedulability for certain intervals of possible task execution times. First, the symbolic, zone-based, MC was used. For stopwatch automata it is implemented as an over-approximation in UPPAAL which still suffices for checking of safety properties, e.g. if the deadline violation can never be reached. However, this technique cannot be used to disprove schedulability of the system as resulting traces may possibly be spurious. Second, the statistical model-checking (SMC) technique was used to provide concrete counterexamples witnessing unschedulability of the model in cases where symbolic MC finds a potential deadline violation and cannot conclude on schedulability. The idea of SMC [15, 16] is to run multiple *sample traces* from a model and then use the traces for statistical analysis which, among all, estimates the probability of a property to be satisfied on a random run of a model. The probability estimate comes with some degree of confidence that can be set by the user among a number of other statistical parameters. Several SMC algorithms that require stochastic semantics of the model have been implemented in UPPAAL SMC [17].

Our contribution to the Herschel-Planck case study is to use our proposed under-approximate randomized reachability analysis techniques in hope to witness unschedulability in places where previously not possible. The summary of (un)schedulability of Herschel-Planck that includes the new results is shown in Table B.1. Symbolic MC finds no deadline violation with over-approximate analysis and is able to conclude schedulability for $\frac{\text{BCET}}{\text{WCET}} \geq 90\%$. SMC find a witness of unschedulability for $\frac{\text{BCET}}{\text{WCET}} \leq 71\%$. Finally, our randomized reachability methods are able to further "breach the wall" of undecidable problem by discovering concrete traces proving unschedulability for $\frac{\text{BCET}}{\text{WCET}} \leq 80\%$. Moreover, for the same $\frac{\text{BCET}}{\text{WCET}}$, randomized reachability finds the deadline violation by three orders of magnitude faster than SMC: the case that took 23 hours for SMC now only takes 23 seconds with randomized methods.

To further verify the proposed efficient development process, we look at several different models of the *Gossiping Girls* problem made by the Master's thesis students – future model developers – and explore the potential of our randomized method. We also perform experiments on a range of other (timed and stopwatch automata) models and compare the performance of our randomized reachability analysis in safety violation detection to that of existing verification techniques of Uppaal: Breadth First Search (BFS), Depth First Search (DFS), Random Depth First Search (RDFS) and SMC. The results are extremely encouraging - randomized reachability methods perform up to several orders of magnitude faster and scale significantly better with increasing model sizes. Furthermore, randomized reachability uses constant memory w.r.t. the size of the model and typically requires only up to 25MB of memory. This is a notable improvement in comparison to the symbolic verification of upscaled and industrial sized models. Each of the experiments in this study was given 16GB of memory.

The main contributions of the paper are:

- A new randomized reachability analysis technique implemented and made available in Uppaal.
- Detection of safety violations up to several orders of magnitude faster than with other existing model-checking techniques.
- Possibility to analyze previously intractable models, including particular settings for the Herschel-Planck case study.
- Searching for *shorter* or *faster* traces with our randomized methods.
- Analysis of strengths and weaknesses of the method based on empirical evidence.

The rest of the paper is structured as follows: In Section 2 we give formal definition of Stopwatch Automata models and in Section 3 we describe the

different randomized methods we tried in this study. In Section 4 we show the user interface of Uppaal. Section 5 gives the experimental setting. Section 6 presents the new results on the Herschel-Planck industrial case study and Section 7 provides more experimental results on other schedulability models. Section 8 demonstrates the efficiency of our randomized method applied on student models of the *Gossiping Girls* problem and Section 9 gives the results on other upscaled models. Finally, Sections 12 and 13 give conclusions and future work.

This STTT journal paper is an extended version of the paper published at FMICS 2021. The major novel sections of this extension in comparison to the original paper are the following (in reading order):

- Section 2 with formal definitions,
- Section 3 with the pseudocode for the randomized reachability algorithm and its respective mentions,
- Section 4 that demonstrates the features of the Uppaal graphical interface w.r.t. our randomized methods,
- Section 5 with experimental setting,
- Section 10 with experiments on research operating system models, and
- Section 11 with discussions about strengths and limitations of our randomized methods.

2 Stopwatch Automata

Timed Automata (TA) are automata extended with real-valued clocks whose values grow uniformly at any state [18]. TA are ideal for describing time-dependent behaviors of systems; however, for preemptive scheduling it is needed to measure the accumulated time the system spends in a certain state. An example would be measuring the progress of a task and stopping it when the task is preempted. To accommodate the need for stopping clocks, an extension that supports derivatives (rates of progression) for clocks being either 1, meaning the clock progresses as per usual, or 0, where the clock is stopped, has been introduced as a Stopwatch Automata (SWA) [13]. Unlike Timed Automata, the reachability analysis of Stopwatch Automata is undecidable. In this section, we present the key definition for Stopwatch Automata based on the formalism from [13].

Let C be a finite set of clocks and V be a finite set of integer variables. Let $u(x)$ define a valuation of $x \in C \cup V$ such that there is a mapping from C to $\mathbb{R}_{\geq 0}$ and from V to $\mathbb{N}$. Let $LC(C, V)$ be a set of linear constraints. A guard $g \in LC(C, V)$ is represented as a finite conjunction of expressions of the form

$c \prec n$, $v \prec n$ or $v \prec c$ where $c \in C$ and $v \in V$, $n \in \mathbb{N}$, and $\prec$ is a relational operator $(<, \leq, >, \geq, =, \neq)$. A set of such *guards* over C and V is denoted as $\mathcal{B}(C, V)$, whereas $\mathcal{P}(C, V)$ is used to denote a powerset. We can change the value of clocks and variables with an *assignment operation* $r(u) \in (\mathcal{P}(C, V))$ where assignments are restricted to be $c = 0$, effectively resetting the clock, and $v = n$, where $c \in C$, $v \in V$ and $n \in \mathbb{N}$.

Definition 12 (Stopwatch Automaton [13])
A Stopwatch Automaton (SWA) $A = (L, l_0, C, V, E, Act, I, D)$ is represented as a tuple where:

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- C is a finite set of clocks that represent time,
- V is a finite set of integer variables,
- $E \subseteq L \times \mathcal{B}(C, V) \times Act \times \mathcal{P}(C, V) \times L$ is a set of edges,
- Act is a finite set of actions,
- $I \colon L \to \mathcal{B}(C, V)$ is a set of location invariants, and
- $D \colon L \times C \mapsto \{0, 1\}$ is a set of rates at which a clock can evolve at a given location.

An edge $e = (l, g, a, r, l') \in E$ represents an edge from location l to location l' with the guard g, action a, and an assignment (reset) r. Semantics of SWA is given in terms of the Timed Transition System that we now define.

Definition 13 (Timed Transition System)
A Timed Transition System (TTS) is a tuple $T = (S, s_0, \Sigma, \to)$ where:

- S is an infinite set of states,
- $s_0 \in S$ is the initial state,
- Σ is a set of labels, and
- $\to \subseteq S \times \Sigma \times \mathbb{R}_{\geq 0} \times S$ is a transition relation. We write $s \xrightarrow{\alpha} s'$ whenever $(s, \alpha, s') \in \to$.

For SWA, a state $s \in S$ is defined as a pair (l, u) with $l \in L$ being a location and u being a valuation over clocks C and variables V. There are two types of transitions: delay and action transitions. Action transitions are the result of following an edge. Delay transitions allow the time to pass and result in the increase of clock valuations such that their valuations after delay d in location l happen w.r.t. the derivative of the clock in the current location defined as $u(c + d) = u(c) + D(l, c) \cdot d$. We now formally define the semantics of SWA.

Definition 14
The semantics of a SWA $A = (L, l_0, C, V, E, Act, I, D)$ is given by a TTS $[\![A]\!]_{sem} = (S, s_0, \Sigma, \rightarrow)$, where $S = L \times u(C, V)$, $s_0 = (l_0, u_0)$, $\Sigma = Act \times \mathbb{R}_{\geq 0}$ and $\rightarrow$ is a transition relation defined as:

- $(s, u') \xrightarrow{a} (s', u')$ iff $\exists (l, g, a, r, l') \in E$, s.t.
  $u \models g$ and $u' = r(u)$ and $u' \models I(l')$
- $(s, u') \xrightarrow{d} (s', u')$ iff $l = l'$ and
  $\forall c \in C(D(l, c) = 0 \Rightarrow u'(c) = u(c))$ and
  $\forall c \in C(D(l, c) = 1 \Rightarrow u'(c) = u(c) + d)$ and $\forall v \in V(u'(v) = u(v))$ and
  $u' \models I(l')$.

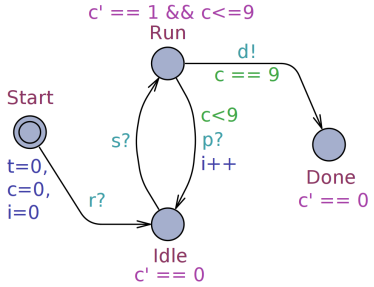


Fig. B.1: Stopwatch Automaton example.

An example of an arbitrary `Task` SWA is shown in Figure B.1 which, behind the scenes, is controlled by some arbitrary *scheduler* (not shown here). `Task` contains a clock `t` that represents the global time, a clock `c` that tracks the total time the automaton occupies CPU for, and a variable `i` that is used to count the amount of times `Task` has been preempted. The automaton consists of four locations - `Start` (initial), `Idle`, `Run` and `Done`. Once `Task` is released by traversing the edge with action `r?`, the location `Idle` is reached where the CPU time clock is paused (`c'==0`). From there, `Task` can be either started (`s?`) and preempted afterwards (`p?`), with the latter action only available if the task has not been completed yet, i.e. ran for less than 9 time units (`c<9`). Each time the task is preempted we increase our preemption counter with `i++`. Note that in UPPAAL clock derivatives are defaulted to 1 for all clocks in all locations, unless specified otherwise. Below we show two example traces for the `Task` automaton:

$$\pi_1 = (\texttt{Start}, t=0, c=0, i=0) \xrightarrow{\texttt{r?}} (\texttt{Idle}, t=0, c=0, i=0) \xrightarrow{\texttt{s?}} (\texttt{Run}, t=0, c=0, i=0)$$
$$\xrightarrow{9} (\texttt{Run}, t=9, c=9, i=0) \xrightarrow{\texttt{d!}} (\texttt{Done}, t=9, c=9, i=0)$$

$$\pi_2 = (\texttt{Start}, t=0, c=0, i=0) \xrightarrow{\texttt{r?}} (\texttt{Idle}, t=0, c=0, i=0) \xrightarrow{7} (\texttt{Idle}, t=7, c=0, i=0)$$
$$\xrightarrow{\texttt{s?}} (\texttt{Run}, t=7, c=0, i=0) \xrightarrow{2} (\texttt{Run}, t=9, c=2, i=0) \xrightarrow{\texttt{p?}} (\texttt{Idle}, t=9, c=2, i=1)$$
$$\xrightarrow{3} (\texttt{Idle}, t=12, c=2, i=1) \xrightarrow{\texttt{s?}} (\texttt{Run}, t=12, c=2, i=1) \xrightarrow{6} (\texttt{Run}, t=18, c=8, i=1$$
$$\xrightarrow{\texttt{p?}} (\texttt{Idle}, t=18, c=8, i=2) \xrightarrow{9} (\texttt{Idle}, t=27, c=8, i=2) \xrightarrow{\texttt{s?}} (\texttt{Run}, t=27, c=8, i=2)$$
$$\xrightarrow{1} (\texttt{Run}, t=28, c=9, i=2) \xrightarrow{\texttt{d!}} (\texttt{Done}, t=28, c=9, i=2)$$

Among the infinitely many traces that reach location `Done`, π_1 has the minimum total time, that is equal to the CPU time, and does not get preempted. In practice, a number of SWA are usually *composed* (executed in parallel) and altogether function as a single system. For simplicity we skip formal definition of *composition* as it depends on the exact model types and extensions used. We refer the interested reader to [3, 16, 19] for more details.

3 Randomized Reachability Analysis

Algorithm 2 Randomized Reachability

```
1:  function RRA(maxSteps, s_0)
2:      steps ← 2^4
3:      while within time budget do
4:          s ← DoRandomWalk(s_0, steps)
5:          if s is terminal then
6:              return concrete trace
7:          steps ← min(steps * 2, maxSteps)
8:  function DoRandomWalk(s, steps)
9:      i ← 0
10:     while i < steps and s is non-terminal do
11:         t ← SelectTransition(s)
12:         d ← SelectDelay(t)
13:         s ← DoDelay(d)
14:         s ← FireTransition(t)
15:         i ← i + 1
16:     return s
```

The purpose of the randomized methods is to explore the state space quickly and be less affected by the state space explosion. The general pseu-

docode for the randomized reachability analysis is given in Algorithm 2. The method is based on a repeated execution of concrete state-based *random walks* through the system. Each random walk is quick and lightweight as it avoids expensive computations of symbolic zone-based abstractions. Moreover, to preserve memory our method does not store any information about already visited states except for the trace of the currently executed random walk. If the target state is found, the concrete trace (e.g. such as trace π_1 from Section 2) is returned (line 6); otherwise, the memory is released before a new random walk is issued. The starting depth of the random walk (line 2) is chosen arbitrarily as a sufficiently small number of steps that is reasonable to explore in the model.

The flaw of such an analysis is its under-approximate nature of exploration which does not allow to conclude on reachability if the target state has never been found. However, the results of [1] hint that randomized reachability analysis has a potential to provide substantial performance improvements in comparison to existing model-checking techniques.

An already existing method – SMC – tries to give valid statistical predictions based on the stochastic semantics. SMC is very similar to the randomized method as it performs cheap, non-exhaustive simulations of the model. In cases where symbolic model-checking techniques of Uppaal are expensive or even inconclusive (for stopwatch automata), SMC is often used as a remedy to provide concrete traces to target states. The stochastic semantics SMC operates on allows for a model to mimic the behavior of a real system; however, this may not be efficient for reachability checking. Consider the timed automaton model in Figure B.2 with the `Goal` location representing the target state we want to discover. The guard `x<=1` on the edge leading to `Goal` requires clock `x` to be at most of 1 time unit. According to the stochastic semantics, at the starting location `Init` SMC would select a delay uniformly in range $[0, 1000]$, which is bounded by the invariant `x<=1000`. This leaves a probability of $\frac{1}{1000}$ to discover `Goal` in 1 step; Alternatively, the "loop" edge is taken which resets clock `x` with the update `x=0` thus resetting all the progress back to the initial state.

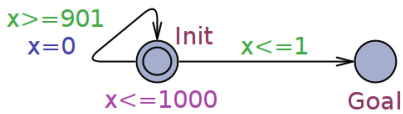


Fig. B.2: Timed Automaton model with a `Goal` target state.

We aim to improve the efficiency of detecting safety violations with our new randomized method by experimenting with several different randomized heuristics and examining their efficiency through an extensive experimental evaluation. A heuristic in this case dictates how a random walk is

Table B.2: Randomized reachability analysis heuristics.

Acronym	Name	Origin	Status
SEM	Semantic exploration	New	Implemented in UPPAAL
RET	Random Enabled Transition	[1]	Implemented in UPPAAL
RLC	Random Least Coverage	New	Implemented in UPPAAL
RLC-A	Random Least Coverage Accumulative	New	Implemented in UPPAAL

performed, i.e. how delays and transitions are chosen. The summary of the heuristics and their status is given in Table B.2. We emphasize attention on the fact that in our algorithm the order in which delays and transitions are selected is reversed from that of SMC: we require selecting a *target transition* first (line 11). The exact delay is then chosen only from that target transition's range of available delays (line 12). Selecting a transition first makes exploration of the state space more uniform and removes a bias towards transitions with larger availability range. The mechanism for choosing delays is common between the heuristics presented below and will be described later in this section. We now explain each heuristic in detail.

SEM An intuitive heuristic we tried, denoted as SEM, is based on the natural semantic exploration of the system. Note that this heuristic is an exception from the delay and transition selection order that was proposed earlier: here, similarly to SMC, delay is selected first, meaning that lines 11 and 12 are switched. In SEM, a meaningful delay, i.e. a delay that leads to an enabled transition, is selected uniformly at random and then a transition is picked uniformly from those available after the chosen delay has been made. In the model from Figure B.2, SEM would choose a delay uniformly from two ranges – $[0, 1]$ and $[901, 1000]$, thus having a probability of $\frac{1}{100}$ to reach `Goal` in 1 step. Overall, we believe this heuristic will struggle the most in systems where certain specific delays are required to reach a target state, e.g. delaying exactly the lower or upper bound of the transition's availability range.

The remaining three heuristics differ only in the implementation of the transition selection method (line 11) which we now explain.

RET As a continuation of our work on randomized techniques from [1] we implement them in UPPAAL for both Timed and Stopwatch Automata. The study proposed two different heuristics for selecting a target transition. A heuristic denoted as RET (Random Enabled Transition) selects one of the eventually enabled transitions, i.e. transitions that are either currently enabled or will become such after a delay, uniformly at random. This means that at each step each transition is equally likely to be selected. When used in

the model from Figure B.2, RET would first choose one of the two transitions at random, having a probability of $\frac{1}{2}$ to reach the `Goal` location in 1 step.

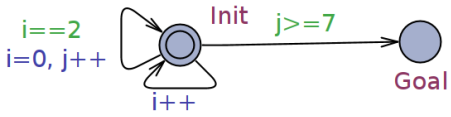


Fig. B.3: Timed Automaton model of a difficult case for the RET heuristic.

RLC, RLC-A Here we introduce a heuristic denoted as RLC that chooses an eventually enabled transition with the *least coverage* for the sending edge, the least coverage being an integer counter that increments every time an edge is traversed. If there is more than one transition with the same least coverage, RLC picks one uniformly at random. In systems that are cyclic or contain multiple loops, RLC provides a more uniform exploration of the state space which may be useful for some models. Consider the model from Figure B.3 that uses two integer variables `i` and `j`. The only initially available edge is the bottom loop edge at the `Init` location which increments the variable `i` by 1 upon each traversal. Once `i==2`, the leftmost loop edge can be taken, resulting in a reset of `i` and increment of `j` (`i=0,j++`). Crucially, if the variable `i` is incremented above the value 2, the leftmost loop edge becomes permanently unavailable. Hence, to reach `Goal` the leftmost edge has to be taken as soon as it becomes available and at least 7 times (`j>=7`) in one run. Since the coverage of the leftmost edge is always lower, the probability for RLC heuristic to discover `Goal` in 1 random walk is 100% while for RET it is less than 1%. The coverage counters, however, are reset at the start of a random walk, making each subsequent run independent of the previous one. We also experiment with a similar heuristic that does not reset the coverage counters and instead keeps them shared among all of the random walks. We denote such *accumulative* heuristic as RLC-A.

Other randomized methods investigated A number of tokenized heuristics, inspired by [20], have been attempted with the intent of storing a small, fixed number of tokens in a clever way to increase the likelihood of reaching the target state faster. Unfortunately, as no considerable improvements have been observed we decided to exclude these heuristics and leave them as future work.

We have also tried using traces of symbolic MC of Uppaal from verification of the Herschel-Planck model to guide the random walks towards the target state. However, even with the RDFS search strategy, all of the symbolic traces have appeared to be spurious due to the over-approximate analysis of

Table B.3: Delay probability distributions used for RET, RLC and RLC-A.

Sequence	1	2	3	4	5	6	7	8	9	10	11
Lower bound	60%	70%	80%	90%	100%	0%	10%	20%	30%	40%	40%
Uniform	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	20%
Upper bound	40%	30%	20%	10%	0%	100%	90%	80%	70%	60%	40%

stopwatch automata. Hence, we could not gain any useful insights with this approach.

To reduce resource demands for the most expensive operation in a random walk – computation of eventually enabled transitions – an alternative heuristic to RET was used in [1] denoted as RCF (Random Channel First). Instead of computing all eventually enabled transitions, RCF first randomly picks a channel and only computes transitions labeled with that channel. However, during implementation of these techniques in Uppaal it became clear that the RCF does not give performance advantages over RET due to the differences in the underlying data structures of Uppaal and the Java prototype from [1]. Therefore, we got rid of the RCF heuristic.

Choosing delay A naive way of choosing delays – uniformly at random from a given range – is likely to not be very efficient. While in some systems that are either small or not sensitive to specific delay values reaching a target state can be doable, in more complex models such a strategy may not be optimal. In [1] we experimented with a few different strategies for choosing delay values, such as 1) uniformly at random, 2) based on predefined probability distribution and 3) based on changing (adapting) delay probability distributions. The experiments have shown the first strategy to be the least efficient, whereas the third one has shown the most potential. Hence, we reuse the third strategy here with slight modifications for RET, RLC and RLC-A heuristics in the implementation of the SelectDelay method at line 12.

The idea behind the adaptive delay choice algorithm is the following: the delays are drawn in accordance to some predefined delay probability distribution which changes with each unsuccessful random walk. Such distribution, in this case, defines probability for lower bound (LB), upper bound (UB) or the values in between the bounds to be chosen. For example, a distribution of 40% LB/40% UB means that it is equally probable that either LB or UB will be selected as a delay, while leaving 20% chance for intermediate delay to be chosen uniformly at random from the range that excludes the bounds. Table B.3 shows the sequence delay probability distributions used in this study. Upon reaching the last distribution in the sequence, the next random walk starts from the first one.

Previously, the cycle of delay probability distributions did not leave any room for intermediate time delays, considering only LB or UB values. The

downside is that for some systems it means that parts of the state space become unreachable by the algorithm; however, experiments have shown this strategy to be surprisingly efficient. To eliminate the flaw of intermediate delay values never being chosen, here we add a 40% LB/40% UB probability distribution, leaving 20% chance to select an intermediate time value. As a result, a target state, if one exists, will be eventually found in any system.

Random walk depth To explore the state space gradually and reduce the risk of a random walk being stuck in an isolated part of the state space with no target state, we increase the random walk depth dynamically as the exploration continues. Specifically, the first batch of random walks at most can perform 2^4 steps. After the full cycle of delay probability distributions is completed, the random walks in the next cycle have their maximum allowed depth doubled, but no further than 2^{18} steps. This approach is similar to a well-known approach of periodically restarting random walks to increase the performance. Should one have some apriori knowledge of the system, it is also possible to manually set the maximum allowed depth in UPPAAL that is a constant value used for all of the conducted random walks.

Shorter or Faster trace Since our techniques cannot disprove reachability of a target state due to under-approximate analysis, searching for errors in large systems, where symbolic techniques struggle, is one of the main expected applications. To aid the developer in analyzing error traces and fixing systems, we implement an option to search for an optimal trace being the either *shortest*, in the size of steps, or the *fastest*, in the amount of total delay. With either one of these options selected, the algorithm searches for the initial trace and afterwards restricts all subsequent random walks to either the current smallest amount of steps or total delay taken, respectively. The exact delay and action transitions taken are recorded in DoDELAY (line 13) and FIRETRANSITION (line 14) functions, respectively. Every randomized heuristic can be used with the *shortest* or *fastest* option and we refer to those by appending "-S" or "-F", e.g. RET-S.

In symbolic model-checking, searching for an optimal trace requires an exhaustive exploration of the state space. Thus, for larger systems, it often drastically increases time and memory demands up to an extent where it becomes impractical. As opposed to that, our randomized techniques do not require more memory as the old trace is being discarded as soon as the new, more optimal one is discovered. On the down side, being a non-exhaustive technique the randomized search cannot guarantee that any discovered trace is indeed the most optimal, endlessly continuing the search. In UPPAAL we let the user provide a *timeout* value (in seconds) which is defaulted to 300 seconds and used as a budget for the main loop at line 3.

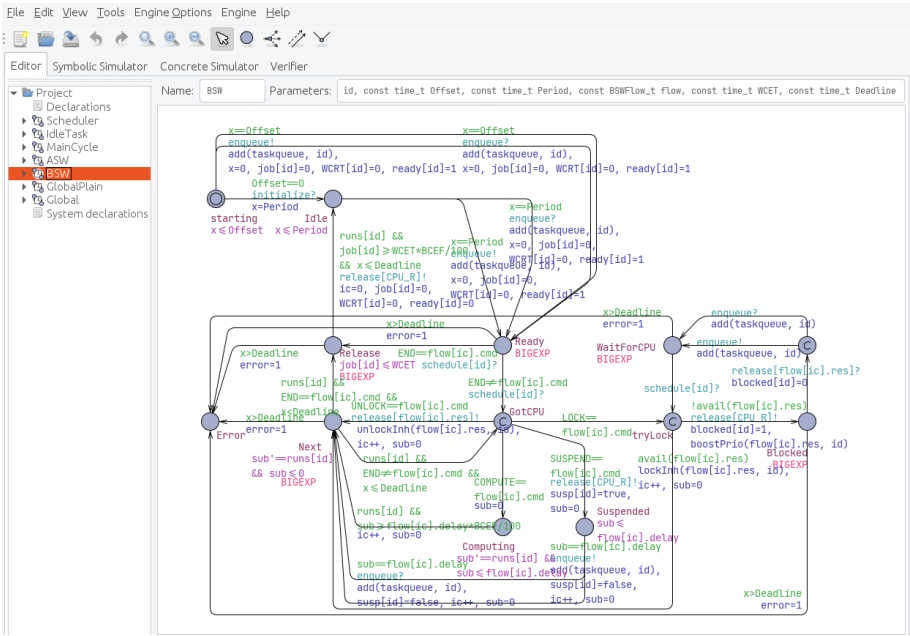


Fig. B.4: The concrete simulator in Uppaal

4 Usage in the tool Uppaal

This section gives an overview of some features of the Uppaal GUI w.r.t. to our randomized methods. Many orthogonal features of the tool are not covered here and we refer the reader to [3] for a comprehensive presentation.

Figure B.4 shows the editor tab of Uppaal. This is the view used to create and edit the model of the system that is going to be analysed. On the left under the `Project` folder one can see all the templates that are part of the current model. In the right pane one of the templates, in this case the task template `BSW` from the Herschel-Planck case study, is open for editing. Finally the model also contains global declarations, declarations local to each template and `System Declarations` that define the composition of templates into the complete system. Apart from the editor tab there are also three other tabs visible in this version of Uppaal. The `Symbolic Simulator` tab will not be shown in this paper as it is not used for the randomized reachability analysis. The two other tabs are explained in the following.

Uppaal supports a subset of the Timed Computation Tree Logic (TCTL) as its specification language. This includes liveness, reachability and leads-to queries. The methods described in this paper supports only reachability queries of the form either *Invariantly p* (`A[] p`) or *Possibly p* (`E<> p`), where

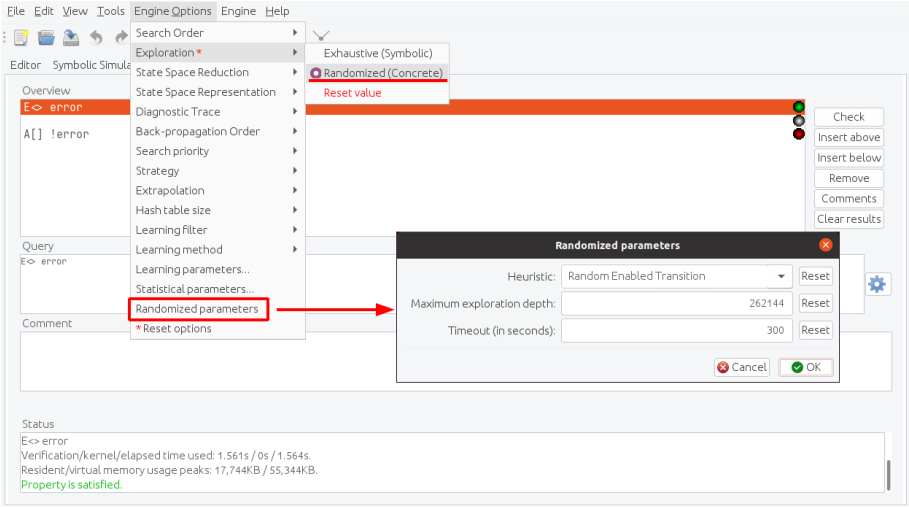


Fig. B.5: The concrete simulator in Uppaal

p is a formula over locations, variables and clocks. For formulas of the type `A[] p` finding a trace represents a proof that the property p does not hold for all states of the system, while for `E<> p` a trace represents a proof that p can be true in some state of the system.

Figure B.5 shows the `Verifier` tab of Uppaal. In the `Engine Options` menu the `Exploration` technique can be selected. By choosing `Randomized (Concrete)` here the techniques from this paper are activated. By selecting `Randomized parameters` the small pop-up shown on top of the screen will appear. The first drop down selects the heuristic to use, while the last field sets a timeout for the random exploration. The middle field sets the maximum depth used for the randomized exploration, with a pre-filled default value of 2^{18}.

Apart from setting these option globally for all queries, it is also possible to set them individually for each query. This makes it possible to operate with a set of queries that can be used as a set of unit test or sanity checks that can be performed quickly after changes to the model.

Figure B.6 show the concrete simulator of Uppaal, where concrete traces obtained from running Algorithm 2 (line 6 can be displayed if the `Diagnostic Trace` option is set to search for *some* trace. There is also a symbolic simulator, but the traces generated by our randomized methods are concrete traces and can as such only be loaded into the concrete simulator. The ability to view the traces in lower left corner of the simulator and to step forwards or backwards through such a trace is important in order to understand the trace and the model. The graphical representation of the model shows the active

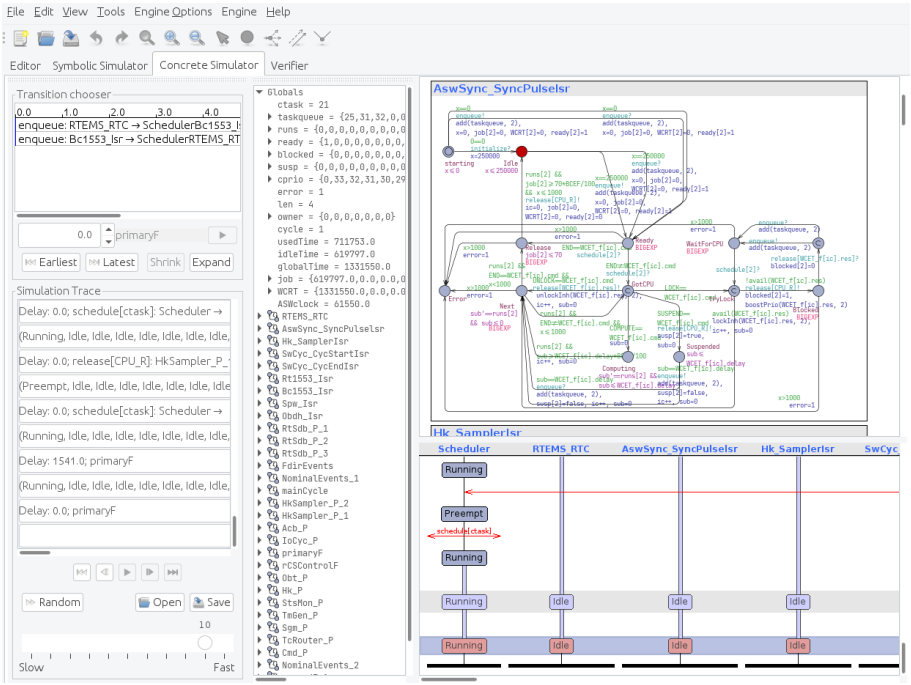


Fig. B.6: The concrete simulator in UPPAAL

locations while the middle pane shows the values of all clocks and other variables. The options for finding shorter or faster traces are especially relevant when trying to manually debug a model by looking at a trace. As an illustrative example we show a trace where `error = 1` for the Herschel-Planck case study, thus indicating that a deadline violation has happened. The message sequence chart in the bottom right corner can help in getting an overview of the communication between different parts of the model.

5 Experimental Setting

All experiments have been conducted on various Timed and Stopwatch Automata models to demonstrate the efficiency of our methods. Sections 6 to 10 give more details about each type of the model. The experiments were ran on a cluster with each instance given 16GB of memory. In tables, we write 'oom' for experiments that ran *out of memory* and 'nf' for cases where concrete witnessing traces were *not found* within the given time. All of the models used in this study can be found online at github, except for the models of an operating system from Section 10.

Symbolic, SMC and randomized methods are available in the newest re-

lease of UPPAAL STRATEGO at `https://uppaal.org/downloads/`. The release comes with the executable and the GUI, the latter of which can be used to open models, change them and perform verification as shown in Section 4.

Reproducibility In order to facilitate reproducibility of experiments, we now explain the procedure of obtaining the experimental results presented in this study. The tables were produced by running the UPPAAL executable from a terminal and measuring the execution time. Since both the SMC and randomized methods may significantly vary in execution times among different replicas of the same experiment, the data shown for these methods in each cell of Tables 4, 6-10 is the average of 100 runs. Therefore, an exact reproduction of the tables is very unlikely. Table 5 is an exception where each cell constitutes a single run. For symbolic, deterministic algorithms we perform only 1 execution.

While the UPPAAL GUI automatically takes care of passing necessary instructions to the engine based on selected options, directly running the engine executable accepts a number of arguments that will determine which methods and under which settings will be executed. The engine executable is named `verifyta` and is contained in the `bin` folder of the UPPAAL release. All of the possible arguments can be viewed in the help menu accessible by running the executable with **-h** argument. The usage of the engine is the following: `verifyta [OPTION]... MODEL QUERY`, where `OPTION` is a space separated list of arguments and `MODEL` is a path to the model file. `QUERY` is an optional argument specifying the path to the query file; however, the necessary query is already specified inside each model file in github and therefore no query files are provided. We now provide the arguments necessary to re-run the experiments from this study.

Selecting the search order (BFS, DFS or RDFS) for symbolic methods is determined by the `-o arg` where `arg` is one of the following:

- `0`: BFS (Breadth First Search) (Default)
- `1`: DFS (Depth First Search)
- `2`: RDFS (Random Depth First Search)

To select the randomized state space exploration it is necessary to specify the exploration type with `--exploration arg` where `arg` is:

- 0: Exhaustive (Symbolic) (Default)
- 1: Randomized (Concrete)

Altering the randomized heuristic is achieved with `--rand-heur arg` where `arg` is:

- 0: RET (Random Enabled Transition) (Default)
- 1: RLC (Random Least Coverage)
- 2: RLC-A (Random Least Coverage Acc.)
- 4: SEM (Naive Semantic Exploration)

The depth of the random walks and the timeout for randomized exploration is controlled with `--rdepth arg` and `--rtimeout arg` arguments, respectively, where `arg` is a positive integer. Finally, to enable generation of the diagnostic trace, pass the `-t arg` argument with `arg` being:

- 0: Some (first trace found)
- 1: Shortest
- 2: Fastest

The procedure of running experiments on different platforms (Linux/MacOS/Windows) is the same in regards to the argument usage. Here are two example commands for running experiments on Linux, assuming the engine executable **verifyta** and the model being in the same folder:

- `./verifyta -o 2 model.xml`
  Runs RDFS on “model.xml”.
- `./verifyta --exploration 1 --rand-heur 4 --rdepth 1000 model.xml`
  Runs SEM with a fixed exploration depth of 1000 on “model.xml”.
- `./verifyta --exploration 1 --rand-heur 0 --rtimeout 3600 -t 1 model.xml`
  Runs RET with 1 hour timeout, searching for the "shortest" trace in “model.xml”.

In order to use SMC methods, it is necessary to verify SMC specific queries [17]. The models with those queries are available at github in the folders with "SMC" prefix. It is also possible to specify options for SMC search (e.g. confidence interval for probability estimation), however in this study we use SMC only for simulation until a single counterexample trace is found and thus no options are required.

Figure B.7 can be reproduced by running RET-S method on the Herschel-Planck model with $f = 75\%$ and a timeout of 20 minutes and plotting the output data from the UPPAAL engine. Each time a shorter trace is found, a corresponding printout is issued by the executable with the new, smaller depth and the time it took to find that trace.

Table B.4: Average time to detect non-schedulability in Herschel-Planck (in seconds). SMC search is limited to 160, 640 or 1280 cycles of 250ms. Each cell shows an average of 100 runs, each with a timeout of 48 hours.

$f(\%)$	SMC(160)	SMC(640)	SMC(1280)	SEM	RET	RLC	RLC-A
68	3378.82	3656.0	2626.11	nf	**14.1**	14.35	14.48
69	6087.64	3258.13	3565.49	nf	15.91	14.32	**13.7**
70	19408.04	16875.89	24322.69	nf	17.59	14.47	**14.77**
71	85837.23	nf	nf	nf	22.54	**16.56**	16.75
72	nf	nf	nf	nf	27.81	**18.42**	18.96
73	nf	nf	nf	nf	31.56	**20.66**	20.68
74	nf	nf	nf	nf	52.53	**38.08**	40.31
75	nf	nf	nf	nf	72.16	**61.98**	68.35
76	nf	nf	nf	nf	**83.12**	328.03	327.32
77	nf	nf	nf	nf	**375.08**	nf	nf
78	nf	nf	nf	nf	**1155.50**	nf	nf
79	nf	nf	nf	nf	**2009.01**	nf	nf
80	nf	nf	nf	nf	**11194.43**	nf	nf
81	nf	nf	nf	nf	nf	nf	nf

6 New Results on Herschel-Planck

According to the previous results on Herschel-Planck model [11], symbolic MC confirmed schedulability for $f = \frac{\text{BCET}}{\text{WCET}} \geq 90\%$. However, symbolic MC cannot be used for disproving schedulability due to over-approximate analysis of automata with stopwatches, used to encode preemption. Thus, SMC was used to generate concrete counterexamples, disproving schedulability for $f \leq 71\%$. For the rest of $f \in (71\%, 90\%)$ both symbolic and statistical MC were inconclusive either due to over-approximation or due to burden in computation time, respectively.

In our experiments we compare SMC to our randomized reachability analysis techniques in an attempt to detect non-schedulability in the Herschel-Planck model with varying execution times in the interval of $[f \cdot \text{WCET}, \text{WCET}]$. The results are shown in Table B.4 with each test case given 48 hours. As the f value gets higher we see the expected growth in computational demands with $f = 71\%$ requiring just under 24 hours for SMC to disprove schedulability, confirming results of [11]. On the other hand, 3 out of 4 of our randomized heuristics were able to detect an error for the same setting of $f = 71\%$ in less than 23 seconds, improving on performance of SMC by three orders of magnitude. Furthermore, the RET heuristic appeared to give the best results, witnessing unschedulability for values of f up to and including 80%. We have also tried running longer experiments of up to 7 days for $f = 81\%$, but no errors were discovered which hints at the possibility of the

Table B.5: Trace length comparison.

f(%)	RET	RET-S	Timeout
68	6882	560	1h
69	7619	568	1h
70	8285	572	1h
71	10411	570	1h
72	12394	571	1h
73	15937	578	1h
74	26605	1549	1h
75	41003	1546	1h
76	40154	1529	1h
77	97258	1536	1h
78	119939	1540	5h
79	129387	1536	5h
80	145493	6455	20h

Herschel-Planck system being schedulable for $f > 80\%$. The SEM heuristic turned out to be the least efficient one, failing to discover any errors, which is likely due to the exponentially small probability of hitting the "right" time windows with the chosen delays. Overall, these experiments showcase the strength of the randomized reachability analysis being fit as a part of an efficient development process that speeds up falsification of models.

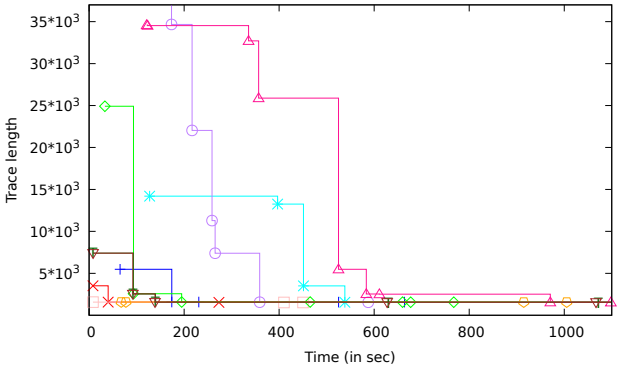


Fig. B.7: 10 runs of RET-S for Herschel with $f = 75\%$.

Once a trace leading to an error is discovered, it might be in the interest of a developer to analyze it to find the cause for the error. The trace, however, can be arbitrarily long, especially for larger systems, making its analysis difficult in practice. In our next experiment we look at the average length of traces found for the Herschel-Planck system and compare the RET heuristic from experiments in Table B.4 against the version of RET with the shortest

trace option enabled - RET-S. In order for a non-exhaustive exploration of RET-S to terminate, we specify the timeout value and increase it w.r.t. to the average time required by RET to find an error. The results are shown in Table B.5. With the given timeout, RET-S shortens the length of the trace by a factor of 12 at minimum. Note that for $f \in [75\%, 79\%]$ the length of the shortest discovered trace is approximately the same – just under 1600 – while the effort to discover such trace is roughly proportional to the average time to detect the first trace (as shown in Table B.4).

The exact value of the timeout has to be decided on by the user which may not be an easy parameter to estimate in the setting of randomized and unpredictable exploration. To better understand how RET-S behaves, we plot 10 runs of RET-S for the Herschel-Planck system with $f = 75\%$ in Figure B.7. In average it took 263.14 seconds to find a trace of sub 1600 steps, while the longest run took 970 seconds.

7 More Schedulability

As already stated, application of symbolic techniques to stopwatch models may provide spurious traces due to over-approximate analysis of Uppaal. If the target state in these models is potentially reachable, we can use SMC to generate concrete and exact traces witnessing the reachability of the goal state. However, SMC can only be applied to systems with broadcast channels as required by the stochastic semantics SMC operates on. In stopwatch models that use handshake channels, our randomized methods become the only solution that can perform a more exact reachability analysis.

Table B.6: Average time of 100 runs to find target state in stopwatch automata models, with a timeout of 2 hours for each run. Symbolic MC techniques provide potentially spurious traces.

Model	#loc	BFS	DFS	RDFS	SMC	SEM	RET	RLC	RLC-A
IMAOptim-0	88	0.09	0.1	0.07	**0.04**	0.07	0.1	0.1	0.08
IMAOptim-1	88	0.21	0.2	0.08	**0.05**	**0.05**	0.08	0.08	0.06
IMAOptim-2	88	0.21	0.26	0.09	**0.06**	0.08	0.11	0.11	0.1
md5-jop	594	0.25	10.8	6.53	n/a	0.15	0.18	0.18	**0.12**
md5-hvmimp	476	0.41	0.85	0.49	n/a	0.1	0.14	0.14	**0.09**
md5-hvmexp	11901	oom	oom	oom	n/a	14.17	19.85	20.18	**8.71**
MP-jop	371	0.39	0.14	0.12	n/a	**0.08**	0.12	0.12	0.09
MP-hvmimp	371	0.35	0.14	0.12	n/a	**0.08**	0.12	0.12	0.09
MP-hvmexp	4388	oom	oom	oom	n/a	13.49	22.95	21.99	**8.59**
simplerts-opt	409	oom	oom	oom	n/a	2.43	**1.48**	nf	nf

We consider more schedulability systems modelled as stopwatch automata. Table B.6 shows experiments for two different sets of schedulability prob-

lems: ARINC-653 partition scheduling of integrated modular avionics systems [21] (denoted as IMAOptim) and Java bytecode systems, originating from TetaSARTS project [22], that are encoded as networks of automata and represent the original layered structure of Java bytecode systems. Our randomized methods discover the target state within 20 seconds even for a huge system with almost 12 thousands of locations, where other techniques either are not applicable or run out of memory.

8 Gossiping Girls

As claimed earlier, the randomized reachability analysis can serve as a useful tool particularly for an efficient development process. It can be used early in the development, as well as in late stages, for a quick falsification of models, i.e. discovery of safety violations as reachability of error states.

To test the efficiency of our randomized methods and challenge them with different model development styles, we look at models of the same problem created by different developers. Specifically, we consider the *Gossiping Girls* problem, where a number of girls n each know a distinct secret and wish to share it with the rest of the girls. They can do so by calling each other and exchanging either only their initial or all of currently known secrets. The girls are organized as a total graph, allowing them to talk with each other concurrently, but with a maximum of 2 girls per call. Some variations of the problem have specific time constraints on the duration of the call or exhibit a different secret exchange pattern, but all with the same final goal of all the girls discovering all of the secrets. This is a combinatorial problem with each girl having a string of n bits which can at most take 2^n values. For a total of n girls this amounts to a string of n^2 with at most 2^{n^2} values. This makes it an incredibly hard combinatorial problem which, when scaled up, quickly exposes the limits of symbolic model-checking due to the state space explosion problem.

We have gathered 10 models of the Gossiping Girls problem made by Master's thesis students as the final assignment for the course on model-checking at Aalborg University in Denmark. These students represent potential future model developers and we use their model to further experiment on applicability of the randomized methods. The implementation details vary from model to model, including timing constraints and secret exchange patterns. We leave the models unchanged and only scale them up to a certain amount of nodes to challenge both symbolic and randomized methods.

We first experiment on the models scaled up to 8 girls and look for a state with of all the girls having exchanged their secrets, while bounded by a certain global time constraint. The results are shown in Table B.7 where each cell represents the average time of 100 runs, with the timeout of 2 hours

Table B.7: Gossiping Girls with 8 nodes. Each cell represents the average time of 100 runs in seconds, with each run limited to 2 hours. Searching for a state with all secrets shared within a certain time.

Model	BFS	DFS	RDFS	SEM	RET	RLC	RLC-A
Gosgirls-1	oom	oom	697.13	nf	**0.39**	6949.95	nf
Gosgirls-2	oom	oom	**0.02**	nf	0.04	0.04	0.04
Gosgirls-3	oom	oom	44.49	nf	**0.02**	**0.02**	0.09
Gosgirls-4	oom	oom	28.35	nf	**0.03**	0.03	nf
Gosgirls-5	oom	oom	229.98	nf	**0.02**	**0.02**	**0.02**
Gosgirls-6	oom	oom	64.00	nf	**3.71**	167.44	1530.99
Gosgirls-7	oom	oom	55.61	nf	**0.17**	15.16	15.6
Gosgirls-8	oom	oom	13.96	nf	0.04	**0.03**	**0.03**
Gosgirls-9	oom	oom	2.08	nf	0.08	**0.07**	0.08
Gosgirls-10	oom	oom	598.64	nf	**0.24**	1.72	nf

Table B.8: Gossiping Girls with 6 nodes. Each cell represents the average time of 100 runs in seconds, with each run limited to 2 hours. Searching for a particular configuration of secrets known.

Model	BFS	DFS	RDFS	SEM	RET	RLC	RLC-A
Gosgirls-1	16.98	oom	oom	2.17	**1.35**	1.60	0.23
Gosgirls-2	**0.04**	oom	360.43	**0.04**	**0.04**	**0.04**	**0.04**
Gosgirls-3	77.96	oom	oom	nf	1.44	0.19	**0.10**
Gosgirls-4	oom	oom	oom	nf	0.03	**0.02**	nf
Gosgirls-5	oom	oom	oom	nf	**0.02**	**0.02**	**0.02**
Gosgirls-6	oom	244.66	2596.62	**5.92**	7.10	nf	nf
Gosgirls-7	oom	oom	oom	nf	**0.14**	75.44	141.20
Gosgirls-8	32.63	oom	oom	nf	**0.11**	3.24	505.99
Gosgirls-9	oom	oom	199.77	**0.10**	13.04	3.65	2.07
Gosgirls-10	oom	oom	209.36	nf	**0.02**	0.03	0.04

for each run. For 9 out of 10 of the models our randomized heuristic RET shows a massive improvement in performance compared to symbolic methods, whereas in 1 model the performance is on the same level. Since the problem is time constrained, the worst performance is that of SEM heuristic which fails to find the target state due to an inefficient way of selecting delays. Importantly, for some models some of the RDFS runs were "lucky" to discover the target state almost immediately, while other "unlucky" tries instead ran out of memory (oom). The oom attempts of RDFS contribute to the performance by noticeably dragging up the average time to find the goal state. Another important factor is memory: unlike symbolic methods, that are given 16GB of memory, our randomized techniques do not run out of memory as its usage is constant w.r.t to the size of the model and amounts to at most 14MB for any of the heuristics for this set of experiments.

Discovery of the state where all the secrets are known is arguably an easy target as such state will eventually always appear as we traverse the state space. This also explains why RDFS was sometimes "lucky" to detect the searched state before it ran out of memory. We now experiment with searching for a particular configuration of secrets in models with 6 girls and show results in Table B.8. Concretely, we divide the 6 girls into two clusters of 2 and 4 girls, and search for a state where each girl knows all the secrets of the other girls in the same cluster, but none from the other cluster. Such a state occurs less often in the state space and is easy to miss, making it a more challenging problem; hence, only 6 girls are considered. Unlike in the previous experiments, the most efficient symbolic search strategy is different for each individual model due to the variance in model implementations. The randomized methods appear largely superior in almost all cases, with the RET heuristic being the most consistent and efficient across all the models. Note that even for 6 girls in a lot of the cases symbolic techniques still run out of memory, whereas our random methods use less than 15MB.

9 Scalability Experiments

We further investigate the efficiency of our randomized methods on a set of standard Uppaal timed automata models. The models are scaled up in order to challenge both symbolic and randomized techniques and the data are provided in Table B.9. The results are truly impressive – randomized methods perform up to 4 orders of magnitude faster and scale significantly better.

Even though the SEM heuristic shows the best performance on many models, its inefficient way of selecting delays causes it to completely miss target states on some models as demonstrated by all of the experiments in this study. Moreover, due to under-approximation, it is possible to construct "evil" examples for any heuristic, rendering it inefficient. Therefore, we make all of the heuristics available in Uppaal and provide a discussion on strengths and limitations of the randomized reachability analysis in Section 11.

10 Operating System models

To further validate the strengths and weaknesses of our proposed methods, we perform experiments on a large model of a research operating system MCSmartOS [23] which is based on a micro-kernel architecture and provides a set of features to the higher system layers. The Uppaal models for the operating system have been developed by [24] as a network of Stopwatch Automata (to model preemption). The models are limited to include a feature set consisting of preemptive multitasking, priority-driven scheduling, task

Table B.9: Average time of 100 runs in seconds to find target state in Timed Automata within 2 hours per run.

Model	BFS	DFS	RDFS	SEM	RET	RLC	RLC-A
csma-cd-20N	20.2	oom	**0.02**	0.03	0.07	0.06	0.21
csma-cd-22N	37.48	oom	oom	**0.03**	0.08	0.08	0.31
csma-cd-25N	91.0	oom	oom	**0.05**	0.09	0.1	0.55
csma-cd-30N	313.54	oom	oom	**0.05**	0.12	0.19	1.43
csma-cd-50N	oom	oom	oom	**0.46**	0.84	1.19	15.29
Fischer-10N	0.9	22.84	4.3	**0.04**	0.05	1.21	nf
Fischer-15N	8.35	6037.63	9038.96	**0.09**	**0.09**	5.06	nf
Fischer-20N	72.61	oom	oom	0.3	**0.28**	17.28	nf
Fischer-25N	452.45	oom	oom	**0.64**	0.73	36.93	nf
Fischer-50N	oom	oom	90.01	**21.78**	23.79	233.67	nf
FischerME-10N	7.15	0.14	0.02	0.01	0.02	**0.01**	0.02
FischerME-15N	oom	11.45	0.05	0.04	0.04	**0.03**	0.16
FischerME-20N	oom	970.33	0.4	0.11	0.09	0.05	**0.04**
FischerME-25N	oom	oom	83.29	0.25	0.21	0.08	**0.07**
FischerME-50N	oom	oom	174.32	14.87	15.26	**0.49**	4.04
LE-Chan-3N	0.03	0.35	0.04	**0.01**	**0.01**	**0.01**	**0.01**
LE-Chan-4N	oom	oom	107.7	0.95	0.54	4.36	**0.07**
LE-Chan-5N	oom	oom	1167.41	53.21	**31.38**	102.08	nf
LE-Hops-3N	0.02	0.02	0.02	**0.01**	**0.01**	**0.01**	**0.01**
LE-Hops-4N	oom	oom	oom	49.40	**14.57**	428.96	1588.33
LE-Hops-5N	oom	oom	1108.15	63.44	**35.15**	36.49	49.00
Milner-N100	0.45	0.16	2.72	nf	**0.11**	**0.11**	0.12
Milner-N500	44.44	10.56	1619.75	nf	**1.19**	1.2	1.43
Milner-N1000	488.41	110.35	36455.73	nf	**4.44**	4.45	4.59
Train-200N	oom	5.64	6.06	5.91	**5.4**	16699.98	nf
Train-300N	oom	28.19	30.28	**25.62**	26.53	nf	nf
Train-400N	oom	85.22	90.66	**67.91**	70.87	nf	nf
Train-500N	oom	210.89	223.13	**181.99**	188.9	nf	nf
Train-1000N	nf	3461.17	3542.08	**2192.12**	2541.57	nf	nf
Train-2000N	nf	71286.92	oom	**19229.02**	23233.21	nf	nf

synchronization, resource management, and time management. For more details, including the models themselves, we refer the interested reader to the mentioned paper.

Running symbolic and randomized methods produces the results shown in Table B.10. SMC is not applicable due to presence of handshake communication between the components and RLC-ACC is not included in the Table as it produced no results in given time. For larger models, e.g. with 5 tasks and 3 resources (5T3R), randomized methods sometimes fail to discover the target state within the time budget (denoted with italic font); such runs are excluded from the average, indicating only the potential best-case average. Overall, the performance of our randomized methods is significantly worse

Table B.10: Average time of 100 runs in seconds to find the target state in SWA models of the MCSmartOS research operating system with 1 hour timeout per run. The numbers in *italic* represent cases where some of the experiments have not produced any results (within 1 hour) and are excluded from that average. The suffix -1k represents randomized methods limited to 1000 depth. Symbolic MC techniques provide potentially spurious traces.

Model	BFS	DFS	RDFS	SEM	RET	RLC	SEM-1k	RET-1k	RLC-1k
OS-4T2R-Q1	3.75	4.13	**0.81**	492.77	326.49	1,161.65	4.35	3.47	9.54
OS-4T2R-Q2	13.02	4.14	**0.46**	160.85	184.23	280.97	4.20	1.85	1.99
OS-4T2R-Q3	12.90	3.99	**0.45**	222.72	158.55	250.33	2.91	1.46	1.54
OS-4T2R-Q4	4.34	3.25	**0.27**	54.34	45.49	89.87	1.11	0.72	0.70
OS-5T3R-Q1	63.32	550.49	**33.12**	*1,571.20*	*1,428.50*	*1,273.75*	69.47	49.69	85.56
OS-5T3R-Q2	337.89	550.74	**20.58**	*1,708.53*	*1,360.02*	*1,824.89*	99.94	47.53	79.29
OS-5T3R-Q3	337.16	368.22	**18.08**	*1,594.32*	*1,575.52*	*1,370.53*	94.31	42.90	76.22
OS-5T3R-Q4	73.08	157.79	**8.63**	*942.57*	*1,074.10*	*1,314.64*	28.81	11.85	27.68

than that of symbolic methods, especially of a clearly dominant RDFS. We assume two potential reasons for that: (1) spurious traces of SWA automata that are found quickly, but do not exist in the actual state space, and (2) the randomized methods stumbling upon *combination locks* that we describe in Section 11. In the first case (1) and similarly to Section 7, our methods yield concrete and non-spurious traces, guaranteeing their existence in the model. To improve against potential combination locks (2), we perform experiments with randomized methods limited to a predefined depth of a 1000, denoting each heuristic with a suffix **-1k**. This greatly increases performance and now all experiments find the goal state. However, the performance is still worse than that of symbolic counterparts. Randomized methods are clearly not a panacea and we now discuss their limitations.

11 Strengths and limitations

In this section we elaborate on limitations and strengths of our randomized reachability analysis that we have observed during various phases of experiments. Note that the list may not be complete, but should give a good idea on expectations when using methods.

"Hitting" exact delay

One of the main weaknesses for all of the presented heuristics are models where a "wrong" delay choice in one state influences the availability of transitions in the following steps. Consider the automaton in Figure B.8 that has two clocks - `x` and `y`. Regardless of the previous delay choices, the maximum possible delay at location `Mid` will always be exactly 1 due to update `y=0` and invariant `y<=1`. Since the transition leading to the `Goal` requires clock `x`

$\in [3;4]$ (`x>=3 && x <= 4`), it is clear that the delay choice made at location `Init` must be in range $[2;4]$ for the `Goal` to be discoverable.

However, none of our heuristics are able to deduce this information before getting to the location `Mid`. The probability for RET, RLC or RLC-ACC to detect `Goal` in a single run then amounts to $\frac{1}{11} \cdot 0.2 \cdot \frac{4-2}{10-0} = 0.364\%$, where the first fraction comes from only 1 out of 11 random walks being allowed to do a uniform delay choice, but only at a 20% probability. This would require an average of 275 random walks to detect the goal. For SEM this probability constitutes 20% as each random walk is guaranteed to perform a uniform choice of delay; nonetheless, with large possible delay value intervals and potentially large number of such choices on the way, the probability to discover the target state can become so small that it will be practically infeasible. In such cases, running symbolic methods with sufficient memory is likely to detect the goal state faster.

This indicates that the way of modelling the problem can severely affect the efficiency of the methods to find concrete witnessing traces. We believe this can be a reason why different models in Table B.8 have been successfully verified by seemingly random methods (both symbolic and concrete), without any of the methods being clearly dominant.

In order to maximize the performance of randomized methods one should focus on creating models such that the invariants on the location would not be able to limit time progression that disallows delaying "enough" to enable an edge. An easy solution for Figure B.8 would be to introduce an extra edge from `Init` to `Mid`, such that the allowed ranges for clock x would be $[0;2)$ on one and $(4;10]$ on the other edge. This would reduce the problem to selecting the right combination of delaying either LB or UB, which is what our randomized heuristics are designed to be good at.

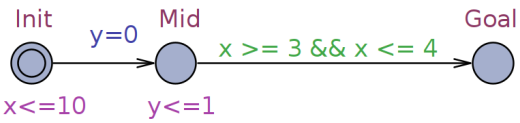


Fig. B.8: Timed Automaton model of a difficult case for any heuristic.

Combination locks

The general disadvantage of randomized and under-approximate methods is their weakness to *combination locks* - cases where only a particular sequence of delays and transitions leads to the target state, while any deviation from that sequence potentially resets the whole progress. Storing no passed states means that the probability of each separate random walk to detect the goal state does not increase over time.

A hard combination lock will typically require not only a consistently accurate choice of delays (as described above) and/or transitions, but also a "correct" configuration of discrete variables, enabling transitions that lead to the target state. Achieving such "correct" configuration of discrete variables can in itself be considered a combination lock, and so on. In such cases, our randomized methods will be easily surpassed by symbolic methods, as long as sufficient amount of memory is provided.

Depth of exploration

Just like with any other method, limiting the depth of exploration significantly reduces resource demands for verification. This is particularly relevant for randomized heuristics as they do not store passed states.

The main advantage of our methods lays in conducting large amounts of state space traversals in a short time, such that even the very unlikely events eventually are discovered. This is likely the main reason why the simplest heuristics (like SEM) often show surprisingly good performance – they are much cheaper and therefore can be executed more times in the same time period.

In large cyclic systems, randomized methods suffer from spending a substantial amount of time in unpromising parts of the state space, e.g. after some "wrong" choice was made that prevents the discovery of the target state. Iterative deepening of the search, described in Section 3, is a partial solution to this problem that lets the model be explored with limited depth of the search before committing to long and demanding random walks. An easy way to improve the performance of randomized methods for large systems is to specify a rough (over-approximated) estimate of the search depth. Such an estimate could come from the depth of previously detected errors during iterative model development. In many of our experiments we have observed the performance benefit of specifying an estimate of the search depth as can be seen in the results of Table B.10. We also note that even though it is (currently) not possible to directly limit the search depth for symbolic methods in Uppaal, one may use a modelling trick e.g. a local step counter or an extra component that halts the model after the desired amount of steps.

Summary

We emphasize that our randomized reachability methods should be used as a supplemental method to symbolic verification and as a convenient tool for quick checking of the model's intended behavior. The advantage of our methods is most prominent for models with a very large state space where the potential benefits of saving time and memory are the highest or where traditional symbolic methods do not terminate.

There are models where detection of target states can be practically infeasible with our methods. However, we believe that such "difficult" models are not frequent in practice and that our non-uniform sampling of the automata language is "guided" towards safety violation states that are often modelled to be at LB or UB of transition availability ranges.

12 Conclusion

We have presented a new method of randomized reachability analysis in the domain of model-based verification. The method excels at detection of safety violation states, by means of quick and lightweight random walks through the system. Randomized reachability analysis explores the state space in an under-approximate manner and can only conclude on reachability if the target state is discovered. However, in many cases this method significantly outperforms other existing techniques at reachability checking. Unfortunately, our randomized methods are not a panacea and for some models reachability checking may be impracticable due to e.g. combination locks. Randomized reachability analysis should therefore be treated as a very useful addition to the process of model development: it provides an efficient way of checking models for potential bugs or violations during the development and can be followed by exhaustive and expensive symbolic verification at the very end. The randomized method also supports the search for either *shorter* or *faster* trace to the target state, which improves the process of debugging the model. The randomized reachability analysis is implemented and made available for use in the model checker Uppaal.

To validate the efficiency of our method, we have performed extensive experiments on models of varying size and origin. The results are extremely encouraging: randomized reachability analysis discovers safety violations up to several orders of magnitude faster. In particular, a case that could previously be analyzed by SMC in 23 hours now only takes 23 seconds. Moreover, our randomized methods discover traces to target states in cases that were previously intractable by any of the existing techniques either due to state space explosion or inconclusiveness in verification of stopwatch models.

13 Future Work

Further investigations into tokenized, coverage-based and guided methods can be done to improve the efficiency of the method. Some combinations of static analysis of the models with either fixed or dynamic look-ahead for the random walk could result in better performance of the method.

One future goal is to perform a more thorough and independent user evaluation of the benefits of the randomized reachability analysis. This could

indicate the need for more parameters to be manually set by the user, such as custom delay probability distribution, or could highlight other areas for improvement of randomized methods.

Even though heuristics like RET aim at the equal probability of traversing transition of a current state, by disregarding "width" of guards, they give no guarantees regarding the uniformity of the exploration with respect to the language inclusion measurement. This, however, has been demonstrated possible by [25] and could be an interesting direction for the future work as to guide the search towards the least explored areas of the state space as a mean of discovering the target states hidden behind combination locks.

Automatic sanity checks is another improvement that can noticeably enhance the user experience and aid during model development. An implementation [26] for Uppaal of such sanity checks has been undertaken as a master thesis project [27] in the Formal Methods & Tools group at University of Twente. This report demonstrates the usefulness of such sanity checks and highlights the need for quick feedback to the tool user. Our randomized method is highly suitable for this purpose.

14 Acknowledgments

This project is supported by the ERC Advanced Grant Project: LASSO: Learning, Analysis, Synthesis and Optimization of Cyber-Physical Systems, and by the Villum Investigator project S4OS: Synthesis of Safe, Small, Secure and Optimal Strategies for Cyber-Phyiscal Systems.

References

[1] A. Kiviriga, K. G. Larsen, and U. Nyman, "Randomized Refinement Checking of Timed I/O Automata," in *Dependable Software Engineering. Theories, Tools, and Applications*, J. Pang and L. Zhang, Eds. Cham: Springer International Publishing, 2020, pp. 70–88.

[2] R. Grosu and S. A. Smolka, "Monte Carlo Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, N. Halbwachs and L. D. Zuck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 271–286.

[3] G. Behrmann, A. David, and K. G. Larsen, "A Tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer, 2004, pp. 200–236.

[4] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 01 1986. [Online]. Available: https://doi.org/10.1093/comjnl/29.5.390

[5] A. Burns, *Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach*. USA: Prentice-Hall, Inc., 1995, p. 225–248.

[6] A. Boudjadar, A. David, J. Kim, K. Larsen, M. Mikučionis, U. Nyman, and A. Skou, "Statistical and exact schedulability analysis of hierarchical scheduling systems," *Science of Computer Programming*, vol. 127, pp. 103–130, May 2016.

[7] ——, "A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling," *Science of Computer Programming*, vol. 113, no. 3, p. 236–260, Dec. 2015.

[8] A. Brekling, M. R. Hansen, and J. Madsen, "Moves — a framework for modelling and verifying embedded systems," in *2009 International Conference on Microelectronics - ICM*, 2009, pp. 149–152.

[9] M. Mikučionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard, "Schedulability analysis using uppaal: Herschel-planck case study," in *Leveraging Applications of Formal Methods, Verification, and Validation*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 175–190.

[10] A. David, J. Illum, K. G. Larsen, and A. Skou, "Model-based framework for schedulability analysis using uppaal 4.1," *Model-based design for embedded systems*, vol. 1, no. 1, pp. 93–119, 2009.

[11] 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*, T. Margaria and B. Steffen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 293–307.

[12] S. Palm, "Herschel-planck acc asw: sizing, timing and schedulability analysis," Tech. rep., Terma A/S, Tech. Rep., 2006.

[13] F. Cassez and K. Larsen, "The impressive power of stopwatches," in *CONCUR 2000 — Concurrency Theory*, C. Palamidessi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 138–152.

[14] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540107000089

[15] K. Sen, M. Viswanathan, and G. Agha, "Statistical Model Checking of Black-Box Probabilistic Systems," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 202–215.

[16] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Runtime Verification*, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 122–135.

[17] 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, pp. 397–415, 2015.

[18] R. Alur and D. Dill, "The theory of timed automata," in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Springer, 1992, pp. 45–73.

[19] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Priced timed automata: Algorithms and applications," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 162–182.

[20] K. Larsen, D. Peled, and S. Sedwards, "Memory-Efficient Tactics for Randomized LTL Model Checking," in *Verified Software. Theories, Tools, and Experiments*, A. Paskevich and T. Wies, Eds. Cham: Springer International Publishing, 2017, pp. 152–169.

[21] Han, Pujie and Zhai, Zhengjun and Nielsen, Brian and Nyman, Ulrik, "Model-based optimization of arinc-653 partition scheduling," *International Journal on Software Tools for Technology Transfer*, Feb 2021. [Online]. Available: https://doi.org/10.1007/s10009-020-00597-6

[22] K. Søe Luckow, T. Bøgholm, and B. Thomsen, "A Flexible Schedulability Analysis Tool for SCJ Programs," http://people.cs.aau.dk/~boegholm/tetasarts/, Accessed: 2021-05-07.

[23] R. Martins Gomes, M. Baunach, and L. Batista Ribeiro, "Mcsmartos: A dependable os for compositional embedded systems," Mar. 2017.

[24] L. Batista Ribeiro, F. Lorber, U. Nyman, K. G. Larsen, and M. Baunach, "A modeling concept for formal verification of os-based compositional software," in *Currently Under Review*, ser. UnderReview'22. New York, NY, USA: Association for Computing Machinery, 2022.

[25] B. Barbot, N. Basset, M. Beunardeau, and M. Kwiatkowska, "Uniform sampling for timed automata with application to language inclusion measurement," in *Quantitative Evaluation of Systems*, G. Agha and B. Van Houdt, Eds. Cham: Springer International Publishing, 2016, pp. 175–190.

[26] R. Onis, "UrPal," https://github.com/utwente-fmt/UrPal, Accessed: 2021-05-18.

[27] R. Onis, "Does your model make sense? : Automatic verification of timed systems," December 2018. [Online]. Available: http://essay.utwente.nl/77031/

Paper C

Monte Carlo Tree Search for Priced Timed Automata

Peter Gjøl Jensen, Andrej Kiviriga, Kim Guldstrand Larsen,
Ulrik Nyman, Adriana Mijačika and Jeppe Høiriis Mortensen

The paper has been published in the
Proceedings of the Quantitative Evaluation of Systems – QEST 2022
LNCS Vol. 13479, pp. 381–398, 2022.

The layout has been revised.

Abstract

Priced timed automata (PTA) were introduced in the early 2000s to allow for generic modelling of resource-consumption problems for systems with real-time constraints. Optimal schedules for allocation of resources may here be recast as optimal reachability problems. In the setting of PTA this problem has been shown decidable and efficient symbolic reachability algorithms have been developed. Moreover, PTA has been successfully applied in a variety of applications. Still, we believe that using techniques from the planning community may provide further improvements. Thus, in this paper we consider exploiting Monte Carlo Tree Search (MCTS), adapting it to problems formulated as PTA reachability problems. We evaluate our approach on a large benchmark set of PTAs modelling either Task graph or Job-shop scheduling problems. We discuss and implement different complete and incomplete exploration policies and study their performance on the benchmark. In addition, we experiment with both well-established and our novel MTCS-based optimizations of PTA and study their impact. We compare our method to the existing symbolic optimal reachability engines for PTAs and demonstrate that our method (1) finds near-optimal plans, and (2) can construct plans for problems infeasible to solve with existing symbolic planners for PTA.

1 Introduction

The world is full of planning and scheduling problems that have impact on the real world. Finding optimal solutions for such problems can be of great importance for profit maximization or resource minimization, affecting financial success and sustainable development. In general such problems do not just have one solution, but many solutions – with varying cost. These scheduling problems are one sub-field within operations research, and lots of effort has been put into finding both optimal and near optimal solutions to them.

One technique that has been successfully applied to planning is that of model checking, e.g. BDD based model checking [1]. For optimal planning problems involving timing constraints, the notion of priced timed automata was introduced in the early 2000s, with initial decidability results [2, 3] based on so-called *corner-point regions* and later with efficient symbolic forward reachability algorithms using so-called *priced zones* made available in the tool Uppaal Cora. Here a generic and highly expressive modeling formalism is provided, extending the classical notion of timed automata [4] with a cost-variable (to be optimized), but also providing support for discrete variables over structured (user-defined) types, as well as user-specified procedures [5]. In fact, the notion of PTA allows for an extension of Planning Domain Definition Language (PDDL) 2.1 at level 3 towards duration-dependent and con-

tinuous effects to be encoded as demonstrated by [6]. Most recently so-called *extrapolation* techniques have been introduced for more efficient analysis of PTA, implemented in the tool TiaMo [7].

Applications of PTA and Uppaal Cora are several and from a variety of areas [8], e.g. power optimization of dataflow applications [9], battery scheduling [10], planning of nano-satelites [11, 12], grape harvest logistic [13], programmable logic controllers [14], smart grids [15], service oriented systems [16], and optimal multicore mapping of spreadsheets [17] to mention a few.

Despite the success of PTA and Uppaal Cora, we still believe that the performance may be improved by exploiting advances made by the planning community. Thus, we consider in this paper various ways of exploiting Monte Carlo Tree Search (MCTS) to further improve performance of PTA optimization. MCTS is a powerful technique that has seen application in many domains requiring (near-) optimal planning, including problem instances where the size of the search-space makes symbolic and complete methods infeasible. In particular, MCTS [18] has already been applied directly to Job-shop [19] scheduling problems. We benchmark our implementations of MCTS based analysis of PTA on Job-shop and Task graph problems and compare against the two tools Uppaal Cora [20] and TiaMo [7].

The rest of the paper is organized as follows: First we formally define Priced Timed Automata, then we introduce a general formalization of Monte Carlo Tree Search along with specific PTA policies. Finally we discuss additional enhancements and present our experimental evaluation.

2 Priced Timed Automata

The priced timed automaton [21] is an extension of timed automaton [4] with prices on both locations and transitions. Delaying in locations entails a price growth based on fixed price (cost) rate, while taking transitions is associated with a fixed price. We now present the formal definition of priced time automaton and its semantics based on [20].

Let $\mathbb{C}$ be a set of clocks. The set of constraints over clocks $\mathbb{C}$, $B(\mathbb{C})$, are defined as the set of conjunctions of atomic constraints of the form $x \bowtie n$, where $x \in \mathbb{C}$, $\bowtie \in \{<, \leq, =, >, \geq\}$ and $n \in \mathbb{N}_{\geq 0}$. Such constraints – guards and invariants – allow to restrict the behavior w.r.t. the values of clocks. The power set of $\mathbb{C}$ is denoted as $2^{(\mathbb{C})}$.

Definition 15 (Priced Timed Automaton)
A Priced Timed Automaton (PTA) over clocks $\mathbb{C}$ and actions Act is represented as a tuple $A = (L, l_0, E, I, P)$ where:

- L is a finite set of locations,

- $l_0 \in L$ is the initial location,
- $E \subseteq L \times \mathcal{B}(\mathbb{C}) \times Act \times 2^{(\mathbb{C})} \times L$ is a set of edges where an edge connects two locations and contains a guard, an action, and a set of clocks to be reset,
- $I\colon L \rightarrow, \mathcal{B}(\mathbb{C})$ is a set of location invariants, and
- $P\colon (L \cup E) \rightarrow \mathbb{N}$ assigns cost rates and cost increments to locations and edges, respectively.

In the case of $(l, g, a, r, l') \in E$, we write $l \xrightarrow{g,a,r} l'$. A clock valuation v over $\mathbb{C}$ is a mapping $v\colon \mathbb{C} \rightarrow \mathbb{R}_{\geq 0}$ and $\mathbb{R}^{\mathbb{C}}$ denotes a set of all clock valuations. The semantics of a PTA is defined in terms of a priced transition system:

Definition 16 (Priced Transition System)
A Priced Transition System (PTS) over actions Act is a tuple $T = (S, s_0, \Sigma, \rightarrow)$ where:

- S is a set of states
- s_0 is an inital state,
- $\Sigma = Act \cup \mathbb{R}_{\geq 0}$ is the set of labels, and
- $\rightarrow \subseteq (S \times \Sigma \times \mathbb{R}_{\geq 0} \times S)$ is a set of labelled and priced transitions. We write $s \xrightarrow{a}_p s'$ whenever $(s, a, p, s') \in \rightarrow$.

Now a PTA $A = (L, l_0, E, I, P)$ defines a PTS $T_A = (S, s_0, \Sigma, \rightarrow)$, where the set of states S are pairs (l, v), with $l \in L$ is a location and v is a clock valuation s.t. the invariant $I(l)$ of l is satisfied by v, denoted $v \models I(l)$.

There are two possible types of transitions between states: *action transitions* and *delay transitions*. Action transitions are the result of following an enabled edge in the PTA A. As a result, the destination location is activated and the clocks in the reset set are set to zero, and the price of the transition is given by the cost of the edge. Formally:

$$(l, v) \xrightarrow{a}_p (l', v') \text{ iff } \exists (l, g, a, r, l') \in E, \text{such that}$$
$$v \models g \wedge v' = v[r] \wedge v' \models I(l) \wedge p = P((l, g, a, r, l'))$$

where $v[r]$ is the valuation given by $v[r](x) = 0$ if $x \in r$ and $v[r](x) = v(x)$ otherwise.

Delay transitions allow the time to pass resulting in an increase of the value of all clocks, but with no change of the location. The cost of a delay transition is the product of the duration of the delay and the cost rate of the active location. Formally:

$$(l, v) \xrightarrow{d}_p (l, v') \text{ iff } v' = v + d \wedge v \models I(l) \wedge v' \models I(l) \wedge p = d \cdot P(l)$$

where $v + d$ is the valuation given by $(v + d)(x) = v(x) + d$ for all x. Finally, the initial state is $s_0 = (l_0, v_0)$, where l_0 is the initial location, and $v_0(x) = 0$ for all clocks x. For networks of priced timed automata we use vectors of locations and the cost rate of a vector is the sum of the cost rates of individual locations.

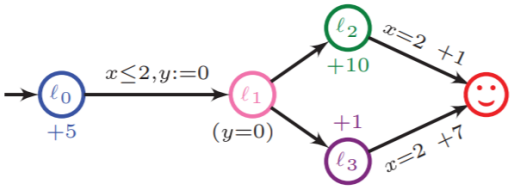


Fig. C.1: Priced Timed Automata example

An example of a PTA is shown in Figure C.1 with clocks x and y and five locations – ℓ_0 (initial), ℓ_1, ℓ_2, ℓ_3, and ℓ_g (goal), with cost rates $P(\ell_0) = +5$, $P(\ell_2) = +10$ and $P(\ell_3) = +1$, and the cost of the edge from ℓ_2 (ℓ_3) to ℓ_g is +1 (+7). Note that the invariant $y = 0$ in ℓ_1 enforces that the location must be left immediately. Below we show two example traces for the automaton:

$$\pi_1 = (\ell_0, x = 0, y = 0) \to_0 (\ell_1, x = 0, y = 0) \to_0 (\ell_3, x = 0, y = 0)$$
$$\xrightarrow{2}_2 (\ell_3, x = 2, y = 2) \to_7 (\ell_g, x = 2, y = 2)$$
$$\pi_2 = (\ell_0, x = 0, y = 0) \xrightarrow{1.5}_{7.5} (\ell_0, x = 1.5, y = 1.5) \to_0 (\ell_1, x = 1.5, y = 0)$$
$$\to_0 (\ell_2, x = 1.5, y = 0) \xrightarrow{0.5}_5 (\ell_2, x = 2, y = 0.5) \to_1 (\ell_g, x = 2, y = 0.5)$$

We see that π_1 reaches ℓ_g with a total cost of $2 + 7 = 9$, whereas the reachability cost of π_2 is $7.5 + 5 + 1 = 13.5$. In fact, among the infinitely many traces that reach ℓ_g, π_1 has the minimum cost. The question of cost-optimal reachability was shown decidable by [2] and later proven to be PSPACE-complete [22]. Here, extending the result for reachability of TAs in [23], it is observed that a PTS semantics with natural-valued delays is complete for PTAs with non-strict guards. Moreover, if k is the maximum constant to which clocks are compared to in guards and invariants, it suffices to consider delays no greater than $k + 1$. In short, in Definition 16 it suffices to consider finite-state PTS with $\Sigma = Act \cup \mathbb{N}_{\leq k+1}$[1] – as in the PTA of Figure C.1, where $k = 2$.

These observations are crucial for our developments of non-symbolic MCTS-based methods for optimal reachability of PTA as we shall see.

[1] $\mathbb{N}_{\leq k+1}$ are all natural numbers less than or equal to $k + 1$.

3 Monte Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a family of algorithms that has been intensely studied in the last decades due to its high success in a range of domains, in particular - game playing. MCTS works on a search tree that grows in asymmetric fashion and in accordance to the results of random samples (or heuristics) that are used to estimate the reward (potential) of the action taken. The tree is iteratively expanded starting from the root node according to four steps:

- Selection: Descend down the tree by selecting the best child according to the chosen policy and until a first unexplored node is met. The selection process typically tries to balance between exploration (visiting promising nodes) and exploitation (visiting nodes with least visits).
- Expansion: Generate a successor of the given state according to the chosen policy.
- Simulation: Estimate the reward of the expanded node by performing simulations, aka roll-outs until the terminal node is reached. Typically, the performance of the algorithm can be drastically improved by a smart simulation strategy.
- Backpropagation: The estimated reward is "backed up" through the tree to update reward estimates.

The first two steps (selection and expansion) are often referred to as *tree policy*, whereas the simulation (roll-out) step is called *default policy*. The algorithm does not have a predefined termination condition and is typically running until either a computational budget (time, memory, etc.) is reached or some different, domain-specific condition is met.

Some of the characteristics that have made MCTS popular in other domains are particularly relevant in the setting of PTA. Tree policy allows one to favor more promising regions of the model which over time leads to *asymmetric* tree growth. This helps alleviate the state-space explosion – the most prominent obstacle in model-checking. Moreover, MCTS being *aheuristic* – easily applicable without the need for domain-specific knowledge – it can be applied to any problem domain as long as it can be modelled as PTA.

We now introduce the formal definition of MCTS and then give the pseudocode of the algorithm – both adapted for the setting of PTA with non-strict guards. Recall that for PTA A with non-strict guards and with maximum constant k (to which clocks are compared) it suffices to consider the *finite* set of labels $\Sigma = Act \cup \mathbb{N}_{\leq k+1}$ to get a finite and complete PTS F_A. We let Σ^* denote the language of finite (natural-valued and bounded) timed strings over Σ and let $\epsilon \in \Sigma^*$ denote the empty string.

By convention we let $|\epsilon| = 0$ and otherwise define $|a_0 \dots a_n| = n$ to be the length of a word. We denote by $w_i \in \Sigma$ the i'th index of the word $w \in \Sigma^*$.

A timed word $w \in \Sigma^*$ of a PTS $T = (S, s_0, \Sigma, \rightarrow)$ is valid iff for $n = |w|$ we have:

$$s_0 \xrightarrow{w_0} s_1 \xrightarrow{w_1} \dots \xrightarrow{w_n} s_{n+1}$$

We let the function $O : \Sigma^* \rightharpoonup S$ denote the outcome of such a valid trace w be $O(w) = s_{n+1}$. By convention we let $O(\epsilon) = s_0$.

Definition 17 (Search Tree)
We define $\Upsilon_T = (N, n_0, \Rightarrow)$ to be the search-tree for a natural- and bounded-valued PTS $T = (S, s_0, \Sigma, \rightarrow)$ as follows:

- $N = \Sigma^*$ is set of nodes,
- $n_0 = \epsilon$ is the root node, and
- $\Rightarrow \subseteq N \times \Sigma \times N$ is the transition relation such that $(n, b, n') \in \Rightarrow$ if and only if $nb = n'$ with $b \in \Sigma$ and $(O(n), b, O(n')) \in \rightarrow$.

We delimit our attention to the most popular MCTS algorithm – the upper confidence bound for trees (UCT) [24]. UCT uses upper confidence bound (UCB1) formula as the tree policy, which addresses the exploration-exploitation dilemma of selecting the most promising paths by treating it as a multiarmed bandit problem. UCB1 makes a good candidate since it is guaranteed to be within a constant factor of the best bound for regret.

Let us define the global functions of the MCTS algorithm. Let $V : N \rightarrow \mathbb{N}$ assign the number of node visits, $Q : N \rightarrow \mathbb{R}$ assign the accumulative reward of the node, $P : N \rightarrow N$ maps to the parent of a node s.t. $P(n) = n'$ where $n' = n\alpha$ and $(n, \alpha, n') \in \Rightarrow$, and $\Upsilon_X : N \rightarrow \mathcal{P}(N)$ defines all children of the node that are valid according to the policy transition relation $\underset{X}{\Rightarrow}$, s.t. $\Upsilon_X(n) = \{n' \mid n \underset{X}{\Rightarrow} n'\}$. The definitions for each policy and respective transition relations are given in the following sections. Children are also partitioned into unexplored (Y^U) and explored (Y^E) ones s.t. $\Upsilon_X(n) = \Upsilon^U_X(n) \cup \Upsilon^E_X(n)$ and $\Upsilon^U_X(n) \cap \Upsilon^E_X(n) = \emptyset$.

Algorithm 3 gives a pseudocode for our PTA-adapted version of the UCT algorithm. The selection strategy used is a standard UCT formula (line 26). The expected reward of a node, determined by the exploitation factor $Q_B \frac{V(n')}{Q(n')}$, is inversely proportional to the average cost found so far which is normalized according to the currently best solution Q_B. The normalization ensures the reward value to be in range between 0 and 1 and thus supports domain (cost range) independence and eliminates the need for any prior knowledge about the reward distribution, which is also apriori unknown for PTAs. The significance of the exploration term is controlled by the value of C constant.

Algorithm 3 The UCT Algorithm. This is a PTA-adapted redefinition of the Algorithm from [18].

function UctSearch(An initial state s_0, a set of goal-states $\mathcal{G}$, an empty set of solved nodes $\mathcal{S}$, an empty set of dead nodes $\mathcal{D}$, and a C_p constant)
  $n_0 \leftarrow s_0$
  while budget remaining **do**
    $n \leftarrow$ TreePolicy(n_0, $\mathcal{G}$, C_p, $\mathcal{S}$, $\mathcal{D}$)
    $\Delta \leftarrow$ DefaultPolicy(n, $\mathcal{G}$)
    BackUp(n, Δ)
    if $O(n) \in \mathcal{G}$ **then**
      MarkSolved($n, \mathcal{S}$)
    if $O(n) \notin \mathcal{G}$ **and** $Y(n) = \emptyset$ **then**
      Prune($n, \mathcal{D}$)
  return BestChild($n_0, 0, \emptyset, \mathcal{D}$)
function TreePolicy(n, $\mathcal{G}$, C_p)
  while $O(n) \notin \mathcal{G}$ **do**
    if $Y_X^U(n) \neq \emptyset$ **then**
      return Expand(n)
    else
      $n \leftarrow$ BestChild($n, C_p, \mathcal{S}, \mathcal{D}$)
  return n
function Expand(n)
  sample $n' \in Y_X^U(n)$
  $V(n') = Q(n') = 0$
  $Y_X^E(n') = \emptyset$
  add n' to $Y_X^E(n)$
  return n'
function BestChild($n, C_p, \mathcal{S}, \mathcal{D}$)
  return $\underset{n' \in Y_X^E(n) \setminus (\mathcal{S} \cup \mathcal{D})}{\operatorname{argmax}} Q_B \frac{V(n')}{Q(n')} + C\sqrt{\frac{\ln V(n)}{V(n')}}$
function DefaultPolicy($n, \mathcal{G}$)
  while $n \notin \mathcal{G}$ **and** within roll-out budget **and**
  $Y_X(n) \neq \emptyset$ **do**
    sample $n' \in Y_X(n)$ uniformly
    $n \leftarrow n'$
  return reward for n
function BackUp(n, *reward*)
  while $n \neq \epsilon$ **do**
    $V(n) \leftarrow V(n) + 1$
    $Q(n) \leftarrow Q(n) + reward$
    $n \leftarrow P(n)$
function MarkSolved($n, \mathcal{S}$)
  while $n \in \mathcal{G}$ **or** $n' \in \mathcal{S}$ for all $n' \in Y_X(n)$ **do**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{n\}$
    $n \leftarrow P(n)$
function Prune($n, \mathcal{D}$)
  if $n \neq \epsilon$ **and** $Y_X(n) = \emptyset$ **then**
    Prune($P(n)$)
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{n\}$

Once a solution is found, we mark the given node terminal to avoid re-exploration (lines 7 and 38-41). As long as the underlying search-tree is complete (determined by the variant of $\underset{X}{\Rightarrow}$) , the algorithm is guaranteed to (eventually) provide an optimal solution given that one exists.

4 General PTA Challenges

Infinite transition sequences:

MCTS algorithms have in large parts been developed for game playing, probabilistic planning or other, typically finite, state-space problems. However, in the setting of PTA, infinite transition sequences are possible, e.g. due to loops in the model. First and foremost it means that traditional roll-outs, directed at reward estimation, might never come to a halt. To overcome this problem we introduce a maximum budget for a roll-out (line 28). An example of the budget is an upper bound on maximum allowed steps that can be done in the default policy before the simulation is terminated.

Reward evaluation:

In turn, capped roll-out length can pose a problem by introducing the need to evaluate non-terminal states. Fortunately, PTA contains all the necessary information needed to evaluate the current cost of any state, including non-terminals. We evaluate and back-propagate the reward regardless of whether the rollout has reached a terminal state.

'Dead' states:

Apart from infinite transition sequences, it is possible to encounter states with no possible successors in PTA. In most MCTS algorithm domains such no successor states are also terminal states; however, it is not necessarily the case for PTA. This is an issue for UCT as it is not equipped to deal with such *dead* states. In UCT, a dead state can be encountered either during expansion or simulations step. For the latter we simply terminate the roll-out upon reaching a dead state (line 29). In case of the former, if UCT expands into a dead state, it must have highest so far expected reward. Simulating from a dead state will not generate any new information, resulting in that state being the best-so-far. To avoid computational overhead, we prune dead states and their parent states from the search tree (lines 9 and 42-45) until no dead states remain in the current branch of the tree.

5 Policies

In MCTS, the structure of the search tree is decided by the unfolding mechanism of the tree policy. The same unfolding strategy is also used during the simulation process of the default policy. In this section we discuss different unfolding strategies that we refer to as *policies*. The specific choice of policy can have a dramatic effect on the performance of MCTS (as we shall demonstrate in the experiments). In particular, for PTA, the search-tree transition function $\Rightarrow$ for the PTA in Figure C.1 would for the state $(\ell_0, x = 0, y = 0)$ contain both the delay-action of 2 time units and the delay-action of 1 time unit (which would be repeatable), leading to the exact same configuration with the same total cost, namely $(\ell_0, x = 2, y = 2)$ at cost 10.

We thus explore both incomplete and complete policies, all restrictions over the full search-tree transition function $\Rightarrow$, with the latter category quarantining the existence of at least one optimal trace. Here, an incomplete policy does not retain the entire search-tree and does not guarantee preservation of an optimal solution. As the first policy, we introduce the Unit Delay Policy.

Definition 18 (Unit Delay Policy)
The transition function $\underset{UDP}{\Rightarrow}$ is given directly by $\underset{UDP}{\Rightarrow} = (N \times (Act \cup \{1\}) \times N) \cap \Rightarrow$.

While the UDP policy streamlines the application of delays, we observe a decreasing probability to pick larger delays. A child node (in tree and default policies) is chosen randomly between all available actions from that state and a delay of a single time unit; consequently, the probability for sequential choice of d unit-delay transitions at state s, i.e. delaying d time units, can be captured as follows:

$$Pr(s, d) = \left(\frac{1}{|Act_s| + 1}\right)^d$$

where $s \in S$, $d \in \mathbb{N}$ and assuming that all actions Act_s are available from state s at all times. If a state has actions that are only valid after a certain amount of time, then those actions are considerably less likely to be explored. We anticipate that such a skewed construction of the tree severely affects the ability of MCTS to find optimal solutions.

To alleviate this, we introduce a *Delay Sampling* policy (DSP) that allows us to choose delays according to a more favorable probability distribution by enforcing a particular structure where delay and action transitions are always alternated. We also use this node layer alternation in the policies following the DSP policy giving a clear cut between transitioning by delay or action. Let $X : S \to \mathcal{P}(\mathbb{N})$ be a function that given a state returns a set of natural-valued delays w.r.t. to location-based constants, which includes the smallest

possible delay, the largest possible delay, and a certain amount of delays from in between the bounds. We include only a subset of possible delays, which is limited to contain at most 100 values and at most 30% of the number of possible values (excluding bounds). The set of possible delays is selected in an attempt to reduce potentially huge branching factor due to delay-actions as to direct the search towards more cost-promising paths. Notice that X may change with each subsequent execution of the algorithm, but will not change during. Formally, DSP is defined as follows.

Definition 19 (Delay Sampling Policy)
The DSP policy $\underset{DSP}{\Rightarrow}$ is defined s.t. if
$(n, \alpha, n') \in \Rightarrow$ then $(n, \alpha, n') \in \underset{DSP}{\Rightarrow}$ iff:

- $n' = na$, $a \in Act$, $n = n''d$, $d \in \mathbb{N}$, or
- $n' = nd$, $d \in X(O(n))$, $a \in Act$ and either $n = \epsilon$ or $n = n''a$.

The policy solves the issue of uneven probability distribution for larger delays. However, it is incomplete in the function X not guaranteeing preservation of key delay values. In addition, we note that the policy still considers a fair degree of delay values (up to 100), quickly leading to a significant degree of branching in the search-tree.

As an alternative, we introduce a policy with the behavior inspired by *Non-lazy schedules* of [25]. The idea behind non-laziness is to avoid unnecessary simultaneous idling of both jobs and corresponding resources. If the resource is available, the job should claim the resource unless some other job can also use it. In the latter case, the first job can be delayed to 'pass' the resource to the second job. We do not give the formal definition of Non-lazy schedules here to maintain readability and refer the interested reader to the mentioned paper for more details.

We introduce our Non-Lazy policy with delays restricted to being either zero, to mimic no delay, or a non-lazy delay, representing the smallest non-zero delay leading to some action becoming enabled, similarly to non-lazy schedules. In comparison to DSP this drastically reduces the breadth of the search tree to at most 2 children and in part alleviates the state-space explosion problem. Let $NLD : S \to \mathcal{P}(\mathbb{N})$ give a set of zero and non-lazy delay, and $A' = \{\alpha \in \Sigma \mid s \overset{\alpha}{\not\to}\}$ be a set of actions that are not immediately enabled from a given state.

$$NLD(s) = \{0 \mid \exists \alpha \in \Sigma \text{ s.t. } s \xrightarrow{\alpha} s'\} \cup \{d' \mid d' = \underset{d \in \mathbb{N}_{>0}}{\textbf{arg min}} \{\exists \alpha \in A' \text{ s.t. } s \xrightarrow{d} s'' \xrightarrow{\alpha} s'\}\}$$

We now give a formal definition of the policy.

Definition 20 (Non Lazy Policy)
The NLP policy $\underset{NLP}{\Rightarrow}$ is defined s.t. if
$(n, \alpha, n') \in \Rightarrow$ then $(n, \alpha, n') \in \underset{NLP}{\Rightarrow}$ iff:

- $n' = na$, $a \in Act$, $n = n''d$, $d \in \mathbb{N}$, or
- $n' = nd$, $d \in NLD(O(n))$, $a \in Act$ and either $n = \epsilon$ or $n = n''a$.

In [25] it is shown that non-lazy schedulers preserve optimal solutions for Job-shop scheduling problems; however, this is not the case for all problems expressible as PTA – implying that the method is incomplete for general PTAs.

Lastly we introduce a policy inspired by *Randomized Reachability Analysis* heuristics from [26]. The idea is to consider action transitions and select delays based on availability range of the chosen action transition. This supports an equal probability distribution to traverse each individual action transition irrespective of its availability range in terms of delays and overall provides a 'fair' exploration. The authors of this heuristics demonstrated its efficiency in finding rare events. We here adapt the idea for finding cost-optimal plans under the heuristic that taking only the smallest possible delay for each transition will often lead to a lower cost.

We now give a formal definition of the Enabled Transition policy. Let $LB : S \times \Sigma \to \mathbb{N}$ give the lower bound of the transition's availability range over the actions of a given PTS. Simply put, LB gives the smallest delay after which a certain action can be taken. Formally:

$$LB(s, \alpha) = \begin{cases} 0 \text{ if } \nexists\, d \in \mathbb{N} \text{ s.t. } s \xrightarrow{d} s' \xrightarrow{\alpha} s'' \\ \underset{d \in \mathbb{N}}{\textbf{arg min}}\ s \xrightarrow{d} s_1 \xrightarrow{\alpha} s_2 \text{ otherwise} \end{cases}$$

Definition 21 (Enabled Transition Policy)
The ETP policy $\underset{ETP}{\Rightarrow}$ is defined s.t. if
$(n, \alpha, n') \in \Rightarrow$ then $(n, \alpha, n') \in \underset{ETP}{\Rightarrow}$ iff:

- $n' = na$, $a \in Act$, $d \in \mathbb{N}$, $n = n''d$, $d = LB(O(n''), a)$, or
- $n' = nd$, $a \in Act$, $d \in \{LB(O(n), a') \mid a' \in Act\}$ and either $n = n''a$ or $n = \epsilon$.

Similarly to NLP, ETP is also an incomplete policy but with more relaxed conditions allowing it to consider all eventually enabled (either now or after delay) actions from a given state.

6 Enhancements

To improve on the performance of the MCTS algorithm, we propose the following modifications over the standard MCTS algorithm presented in Algorithm 3.

Building Rollouts. The standard UCT algorithms uses rollouts to estimate the reward of a node, but strictly in a way s.t. the tree is not expanded, as to preserve memory. We propose to add a rollout to the tree under two conditions: if

1. a roll-out reaches the terminal state, and
2. it does so with the so-far-best cost.

We denote such configuration as BR.

Tree pruning with steps. It can be beneficial to perform a step (advance the root) once 'enough' information has been gathered to ensure near-optimal action choice in the root of the search-tree. Two domain-independent techniques – *Absolute pruning* and *Relative pruning* – have been introduced in [27]. They have shown that the Absolute pruning in fact preserves the optimality of the search tree, but concluded that rather few nodes are actually being pruned due to pruning conditions being too strict. We will thus only study the Relative pruning technique.

We briefly recall the condition for Relative pruning (RP), which is dependent on the tunable parameter μ.

Condition 1 (Relative pruning condition)
Node n_i can be pruned if $\exists j$ such that
$V(n_j) > V(n_i) + \mu$, where $i \in \{1, \cdots, k\}, j \in \{1, \cdots, k\}, i \neq j$ and for all i we have $(n, \alpha, n_i) \in \Rightarrow$ with $\alpha \in \Sigma$.

We also propose a simpler method of pruning based on a constant *stepping value*, i.e. a number of samples required in the current root-node before advancing the root of the tree. We denote this pruning technique Stepping pruning (SP).

7 Experiments

We perform experiments on three benchmarks:

1. Job-shop scheduling[2] problems,

[2] `https://github.com/tamy0612/JSPLIB`

2. Task graph scheduling[3] problems of [28] translated to PTA by [29], and
3. satellite mission scheduling problems [30, 31].

We select 120 Task graph models (of thousands) and use all 162 Job-shop models, and all of the satellite models. The largest Job-shop model contains 100 jobs using 20 machines and the largest Task graph consists of 300 tasks (83 chains) executed on 16 machines. To account for randomness of the MCTS and random-search methods, we report the average of 10 executions. For symbolic methods (which are deterministic) we only conduct one execution. All experiments are limited to 10 minutes and the best found solution is reported (if any). The experiments are conducted on AMD Opteron 6376 processors with frequency-scaling disabled running Debian with a Linux 5.8 kernel and limited to 8 GB of memory (except for experiments with TiaMo which is given sufficient memory).

Solving using PDDL (Planning Domain Definition Language) Planners. As a consequence of our restriction to natural-valued delays, it is possible to compile the PTA models into (classical, deterministic) planning problems and apply well-studied classic planning algorithms. To study this, we convert the Job-shop PTA models to PDDL 2.2 with action costs from PDDL 3.1 and use the Fast Downward[4] planner to find cost-efficient plans. We apply some classical algorithms, e.g. greedy best-first search with the FF heuristic for sub-optimal plans [32] and A* with LM-Cut for optimal plans [33]. However, the so-called grounding phase never terminates within the time and memory limit, even for the smallest Job-shop model consisting of 6 jobs and 6 machines. Scaling down the models further (by gradually removing jobs) reveals that the complexity of the model with 3 jobs already surpasses the capabilities of the planner to find a solution in allotted time. It is well-known that if the parameter-space of the actions in PDDL encoding grows large, which is the case for our models, the state-space suffers from an exponential explosion. We thus refrain from comparing to classical planners in the remainder of this section and leave comparison to more complex planners (e.g. temporal planning algorithms) to future work.

Presentation of results. In our graphs we present the relative performance of a method against *Best Known Solutions* (BKS) which is known for the Job-shop and Task graph problems. A 0% deviation indicates that the BKS was found and a 10% deviation denotes a solution that is 110% of the BKS. We refer to the BKS as the reference value. For all but the last experiment we present the results over both benchmarks in one single plot. Figures C.2-C.9

[3] `https://github.com/marmux/spreadsheets`
[4] `https://www.fast-downward.org/HomePage`

are plotted as “Cactus” or “Survival” plots. The y-axis shows the quality of the solution as "% worse than the BKS" (Fig. C.2-C.7). Each method is sorted individually, resulting in monotonically increasing lines. Therefore, data-points from different methods for a given x-value can be produced by different models, showcasing the general trend of each individual method over the benchmark.

We conduct the following sets of experiments:

- **Building Rollouts** where we construct the search-tree if a terminal node is found during rollout,
- **Impact of Stepping** where we experiment with pruning techniques,
- C_p **Sensitivity** where we vary the exploration constant,
- **Policy Study** where we compare the proposed policies, and
- **Comparison w. Existing Methods** where we compare our best performing method with existing solvers for PTA, and
- a study of the methods on a set of more general PTA models stemming from the domain of satellite mission planning.

Building Rollouts. We initially study the impact of the BR enhancement as any configuration without this enhancement is unable to yield results for a significant portion of the benchmarks. As a representative configuration we here present the results with the NLP policy both with and without the SP pruning and the exploration constant C fixed to $\sqrt{2}$. Other configurations demonstrate a similar tendency. We observe in Figure C.2 that only versions with the BR optimization manage to find a solution to all the instances. In particular, we see that the version without both SP and BR produces no results at all (red line). We witness the effect of BR from the plot and see that the best performing configurations are deviating no more than 30% from the reference. In addition, for roughly 50% of the models, this deviation is less than 5%.

Impact of Stepping. In Figure C.3 we compare different stepping sizes for SP and different upper-bounds number of visits (μ) for RP. We here restrict the reported results to the BR variant of the NLP policy. We observe that SP is highly sensitive to the stepping size and see that the smallest step sizes result in worse performance due to a too rapid progression of the root-node while too high values fail to reduce the search-space to a feasible size. We observe a similar tendency with RP wrt. the sensitivity of the μ-value, albeit to a lesser degree. Importantly we observe that SP (using a stepsize of 500)

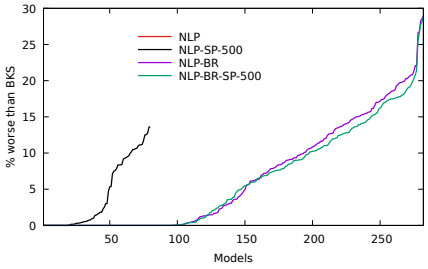


Fig. C.2: The effect of BR and SP on the NLP policy.

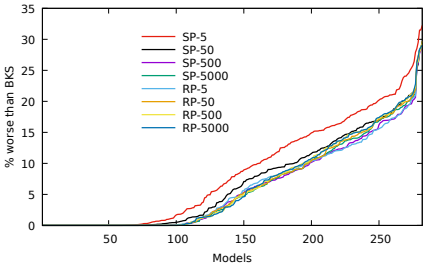


Fig. C.3: Comparison of stepping values for NLP using BR and with $C_p = \sqrt{2}$.

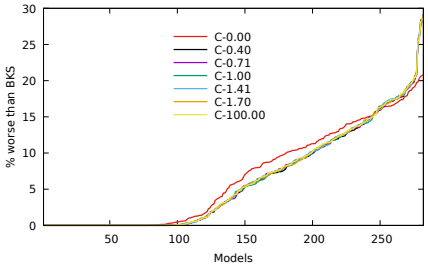


Fig. C.4: Comparison of different C_p values effect on NLP with BR and SP-500 options.

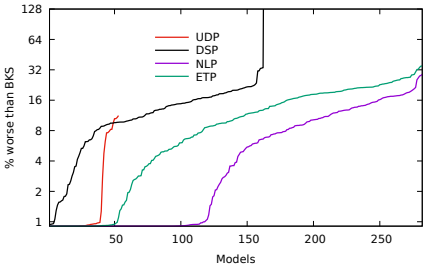


Fig. C.5: Comparison of UDP, DSP, NLP, ETP policies with $C_p = \sqrt{2}$ and the best enhancements used: BR and SP-500.

and RP (with $\mu = 5$) perform similarly well – and we delimit ourselves to reporting only on variants using SP in subsequent experiments.

While using $C_p = \sqrt{2}$ is often considered a good value to strike a balance between exploration and exploitation, we here study the sensitivity to changes in the C_p-value, in particular as our setting is a single-player setting. Specifically we can in Figure C.4 observe the difference in performance when $C_p \in \{0, 0.4, \frac{1}{\sqrt{2}}, 1, \sqrt{2}, 1.70, 100\}$ where the value 100 is chosen arbitrarily as "a sufficiently large value" to force the algorithm to focus purely on exploration. From Figure C.4 we observe that apart from $C_p = 0$, the choice of C_p has little to no impact on the performance – likely due to the fact that our setting is a single player setting. Regarding $C_p = 0$, we conjecture that the effect observed stems from an intensive search around the initially found solution. For instances with a positive effect we believe that a (near-)optimal solution is found within the vicinity of *any* solution, where a negative effect indicate a larger difference between local minima in the search-space. While a small set of models clearly favor $C_p = 0$, we use $C_p = \sqrt{2}$ for the remainder of the experiments as it provides overall good performance and is the value recommended by literature.

Policy Study. The summary on the performance of different policies is shown in Figure C.5. Here we fix the configuration to use the BR and SP enhancements with a step-size of 500. We observe that UDP has the worst performance with less than 20% of problem instances solved within the given time-frame - and significantly worse quality solutions. We believe this to be due to the low probability of selecting larger delays and the state-space explosion of having to consider all possible delays. While DSP is an improvement over UDP, it suffers from a similar problem in that the branching factor can explode leading to a performance degradation. Both NLP and ETP were able to solve all problem instances with near-optimal solutions of at most 28.88% and 35.42% away from the reference value, respectively, however with a clear advantage to NLP.

Comparison w. Existing Methods. Lastly we perform a comparison of our best configuration with other existing state-of-the-art solvers for PTA, namely Uppaal Cora and TiaMo. In addition, we have also adapted the *Randomized Reachability Analysis* (RRA) methods of [26] to search for optimal schedules rather than rare events. We experiment with several of the techniques proposed for RRA (RET, RLC and RLC-A) to search for optimal solutions. We refer the interested reader to the mentioned paper for more details.

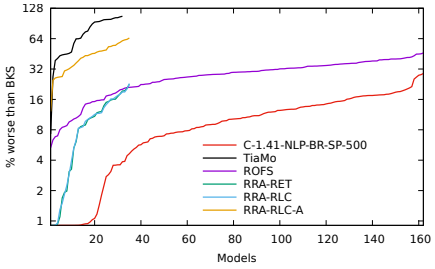


Fig. C.6: Job-shop overview.

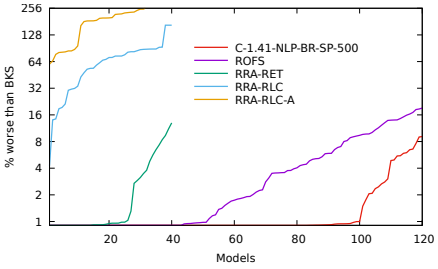


Fig. C.7: Task graph overview.

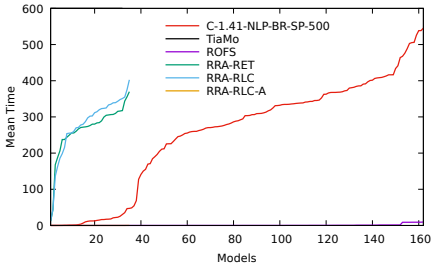


Fig. C.8: Job-shop runtime overview.

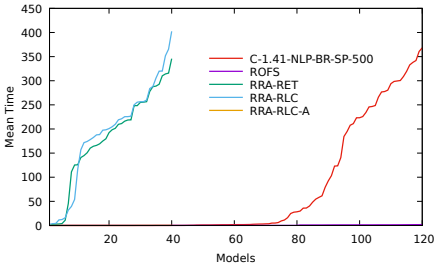


Fig. C.9: Task graph runtime overview.

In the case of Cora we use both the complete and optimal search-method as well as the incomplete *Random Optimal First Search* (ROFS) approach, which allows for a very lightweight search in a depth-first manner while choosing the most optimal action at each step but providing no guarantee wrt. optimality of the returned solution. It is important to note that both Cora (except for the ROFS version) and TiaMo are complete and able to find an optimal solution if given enough time and memory - and that both methods are relying on a symbolic representation of the search-space.

Figure C.6 gives an overview of all the methods for Job-shop scheduling benchmark compared against BKS from [34]. Note that Cora has not managed to solve any instance for either of the benchmarks, primarily limited by the fact that it is a piece of 32bit software only capable of utilizing 4GB of memory. Unfortunately Cora does not provide anytime solutions in its current distribution. Both TiaMo and RRA methods solve less than 20% of the instances, with TiaMo delivering sub-par solutions as it never completes the search within the time-limit, and thus provides only any-time solutions as they are found.

The ROFS algorithm of Cora outperforms both the random search and TiaMo in terms of solved instances, while having a drawback with the quality of the produced plans when compared to our MCTS implementation. In terms of time (Fig. C.8, C.9), the ROFS algorithm is the fastest overall, completing its search within single-digit seconds. We note that the overall quality of the schedules found by ROFS is within a surprisingly reasonable distance from the optimal, indicating that a greedy search strategy is well suited for the given benchmark. We observe the best performance of the proposed MCTS configuration using NLP, SP, BR and $C_p = \sqrt{2}$ and see a deviation of up to 28.88% of the BKS - with a median of deviations of no more than 10.3%. However, investigating the computation time, we can see that the best found solution is in the median produced at 289s and peaking at 546s.

The overview for the Task graph scheduling benchmark compared against BKS from [35] is shown in Figure C.7. Due to its limited support of the PTA syntax, the TiaMo tool was not applicable. For over 80% of the benchmark (100/120 models) the solutions found by NLP are (near-)optimal with the quality of solutions of at most 1% away from BKS. For the rest of the benchmark the performance of NLP slightly worsens reaching at most 9.11% deviation from BKS. In general, the trends for different methods are very similar: RRA methods solve around 33% of models only, while ROFS finds solutions near instantly, but their quality degrades with increased model complexity.

Satellite models. Additionally, we experiment with two satellite cases - GomX-3 and Ulloriaq - designed, delivered, and operated by Danish satellite manufacturer GomSpace. The PTA models for these satellites have been developed in [30] and [31] studies, respectively, and analyzed with Uppaal

Table C.1: Results for different PTA models of satellite problems. MCTS policies executed with $C_p = \sqrt{2}$, BR and SP-500 enhancements enabled. (oom = out of memory)

		DSP	NLP	ETP	ROFS	Cora
gomx3-1day	Mean cost	**186,007** (±0.00%)	188,408 (±1.95%)	**186,007** (±0.00%)	198,292 (±0.00%)	**186,007**
	Time	40.2	49.8	61.5	**0.05**	5.12
gomx3-2day	Mean cost	442,190 (±0.04%)	442,218 (±0.01%)	**442,080** (±0.06%)	478,002 (±0.17%)	oom
	Time	223.3	268.0	230.7	**0.05**	-
5sat	Mean cost	5,072,861 (±4.12%)	5,961,014 (±1.72%)	**3,548,824** (±0.77%)	3,739,730 (±1.82%)	oom
	Time	267.5	366.7	295.8	**0.25**	-
10sat	Mean cost	**5,632,414** (±0.57%)	6,130,961 (±0.68%)	nf	5,687,131 (±2.47%)	oom
	Time	232.4	232.6	600.0	**0.56**	-
MaxData626	Mean cost	nf	nf	nf	**7,458,522** (±2.17%)	oom
	Time	600.0	600.0	600.0	**0.62**	-

Cora (including ROFS). We show the results in Table C.1, but exclude UDP as it produces no results within the time limit. For all models (but one) MCTS provides the best mean cost across all the methods; however, ROFS finds solutions up to 4 orders of magnitude faster and with a modest reduction of quality (up to 10% from the best MCTS method). We believe this is due to a generally small variance in the quality of solutions in the solution-space and the fact that ROFS performs only a single traversal of the model, immediately reporting the result upon reaching the terminal state. For "MaxData626" model MCTS methods timeout without a solution. Further experiments with an increased time-limit of 5 hours do not yield additional results indicating issues with the incompleteness of the methods rather than missing computation-time. The relative efficiency of the ROFS method demonstrates a potential for extending the MCTS method in the direction of a symbolic search, allowing for an efficient and complete MCTS tree-search method, and overcoming the current limitations of the discretized equivalents studied in this paper.

8 Conclusion

We have adapted the Monte Carlo Tree Search (MCTS) algorithm for the setting of problems described as Priced Timed Automata (PTA) – a formalism that can capture the behavior of a wide range of optimization problems such as resource-consumption or -allocation problems. PTA is a very versatile modeling formalism, facilitating more direct modeling of a problem domain.

We introduced a number of complete and non-complete policies that act as unfolding mechanism and decide the structure of the tree. Some domain-independent enhancements to improve the performance and coverage of the algorithm are suggested.

We have evaluated the performance of our MCTS algorithm adapted to PTA on three benchmarks of Job-shop, Task graph and satellite mission scheduling problems and compared it against other state-of-the-art methods and tools. For the first two benchmarks, the results indicate that MCTS is able to find near-optimal solutions for all investigated problem instances. In general, we observed an up to 28.88% and 9.11% deviation (on average) from the best known solution in a set of Job-shop and Task graph scheduling problems, respectively. For satellite models, MCTS methods have found the best cost across all tested methods except for one model where only ROFS was able to produce results, hinting at issues with the incompleteness of MCTS methods.

All this suggests that MCTS is a promising alternative that copes well with the state-space explosion problem where other existing, exhaustive and complete methods perform poorly or fail. We note that the Random Optimal First Search strategy of the tool Uppaal Cora performs well, even when compared to MCTS. The study of more symbolic approaches to MCTS for PTA is left as future work.

Data availability. A reproducibility artifact, which contains binaries, models and scripts to reproduce results can be found at:
`https://doi.org/10.6084/m9.figshare.19772926`

References

[1] S. Edelkamp, "Heuristic Search Planning with BDDs," in *PuK2000*, 2000. [Online]. Available: http://www.puk-workshop.de/puk2000/papers/edelkamp.pdf

[2] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager, "Minimum-Cost Reachability for Priced Time Automata," in *HSCC 2001*, M. D. Di Benedetto and A. Sangiovanni-Vincentelli, Eds. Springer, 2001, pp. 147–161. [Online]. Available: https://doi.org/10.1007/3-540-45351-2_15

[3] R. Alur, S. La Torre, and G. J. Pappas, "Optimal Paths in Weighted Timed Automata," in *HSCC 2001*, M. D. Di Benedetto and A. Sangiovanni-Vincentelli, Eds. Springer, 2001, pp. 49–62.

[4] R. Alur and D. Dill, "The theory of timed automata," in *Real-Time: Theory in Practice*, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds. Springer, 1992, pp. 45–73.

[5] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Optimal scheduling using priced timed automata," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 4, p. 34–40, mar 2005.

[6] H. Dirks, "Finding Optimal Plans for Domains with Restricted Continuous Effects with UPPAAL CORA," ser. ICAPS 2005. American Association for Artificial Intelligence, 2005.

[7] P. Bouyer, M. Colange, and N. Markey, "Symbolic Optimal Reachability in Weighted Timed Automata," in *CAV 2016*, S. Chaudhuri and A. Farzan, Eds. Springer, 2016, pp. 513–530.

[8] P. Bouyer, U. Fahrenberg, K. G. Larsen, and N. Markey, "Quantitative analysis of real-time systems using priced timed automata," *Commun. ACM*, vol. 54, no. 9, pp. 78–87, 2011.

[9] W. Ahmad, P. K. F. Hölzenspies, M. Stoelinga, and J. van de Pol, "Green Computing: Power Optimisation of VFI-Based Real-Time Multiprocessor Dataflow Applications," in *DSD 2015*. IEEE Computer Society, 2015, pp. 271–275.

[10] M. R. Jongerden, B. R. Haverkort, H. C. Bohnenkamp, and J. Katoen, "Maximizing system lifetime by battery scheduling," in *IEEE/IFIP Int. Conf. DSN 2009*. IEEE Computer Society, 2009, pp. 63–72.

[11] H. Hermanns, J. Krcál, and G. Nies, "How Is Your Satellite Doing? Battery Kinetics with Recharging and Uncertainty," *Leibniz Trans. Embed. Syst.*, vol. 4, no. 1, pp. 04:1–04:28, 2017.

[12] A. Korvell and K. Degn, "Designing a Tool-Chain for Generating Battery-Aware Contact Plans Using UPPAAL." Aalborg University, Master Thesis, 2019.

[13] R. Saddem-Yagoubi, O. Naud, K. Godary-Dejean, and D. Crestani, "Model-Checking precision agriculture logistics: the case of the differential harvest," in *Discrete Event Systems*. Springer, 2020.

[14] A. Vulgarakis and A. Čaušević, "Applying REMES behavioral modeling to PLC systems," in *2009 XXII International Symposium on Information, Communication and Automation Technologies*. IEEE, 2009, pp. 1–8.

[15] N. Geuze, "Energy management in smart grids using timed automata," Master's thesis, University of Twente, 2019.

[16] A. Čaušević, C. Seceleanu, and P. Pettersson, "Checking correctness of services modeled as priced timed automata," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2012, pp. 308–322.

[17] T. Bøgholm, K. G. Larsen, M. Muñiz, B. Thomsen, and L. L. Thomsen, *Analyzing Spreadsheets for Parallel Execution via Model Checking*. Cham: Springer International Publishing, 2019, pp. 27–35. [Online]. Available: https://doi.org/10.1007/978-3-030-22348-9_3

[18] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[19] A. Banharnsakun, B. Sirinaovakul, and T. Achalakul, "Job Shop Scheduling with the Best-so-far ABC," *Engineering Applications of Artificial Intelligence*, vol. 25, no. 3, pp. 583–593, 2012.

[20] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Priced Timed Automata: Algorithms and Applications," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Springer, 2005, pp. 162–182.

[21] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn, "Efficient Guiding Towards Cost-Optimality in UPPAAL," in *TACAS 21*, T. Margaria and W. Yi, Eds. Springer, 2001, pp. 174–188.

[22] P. Bouyer, T. Brihaye, V. Bruyère, and J. Raskin, "On the optimal reachability problem of weighted timed automata," *Formal Methods Syst. Des.*, vol. 31, no. 2, pp. 135–175, 2007.

[23] 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*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 125–141. [Online]. Available: https://doi.org/10.1007/3-540-48153-2_11

[24] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[25] Y. Abdeddaïm and O. Maler, "Job-Shop Scheduling Using Timed Automata?" in *CAV 2001*, G. Berry, H. Comon, and A. Finkel, Eds. Springer, 2001, pp. 478–492.

[26] A. Kiviriga, K. G. Larsen, and U. Nyman, "Randomized Reachability Analysis in Uppaal: Fast Error Detection in Timed Systems," in *FMICS 2021*, A. Lluch Lafuente and A. Mavridou, Eds. Springer, 2021, pp. 149–166.

[27] J. Huang, Z. Liu, B. Lu, and F. Xiao, "Pruning in uct algorithm," in *2010 International Conference on Technologies and Applications of Artificial Intelligence*, 2010, pp. 177–181.

[28] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.

[29] A. Ejsing, M. Jensen, M. Muñiz, J. Nørhave, and L. Rechter, "Near Optimal Task Graph Scheduling with Priced Timed Automata and Priced Timed Markov Decision Processes," 2020.

[30] M. Bisgaard, D. Gerhardt, H. Hermanns, J. Krčál, G. Nies, and M. Stenger, "Battery-aware scheduling in low orbit: the GomX–3 case," *Formal Aspects of Computing*, vol. 31, no. 2, pp. 261–285, 2019.

[31] A. Kørvell and K. Degn, "Designing a Tool-Chain For Generating Battery-Aware Contact Plans Using UPPAAL," 2019.

[32] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[33] M. Helmert and C. Domshlak, "Landmarks, critical paths and abstractions: what's the difference anyway?" in *Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

[34] A. Jain and S. Meeran, "Deterministic job-shop scheduling: Past, present and future," *European Journal of Operational Research*, vol. 113, no. 2, pp. 390–434, 1999.

[35] K. Laboratory. Standard task graph set. [Online]. Available: https://www.kasahara.cs.waseda.ac.jp/schedule/index.html

Paper D

Usage-aware Falsification for Cyber-Physical Systems

Andrej Kiviriga, Kim Guldstrand Larsen, Dejan Nickovic and Ulrik Nyman

The paper has been submitted to the
14th ACM/IEEE International Conference on Cyber-Physical Systems, 2023.

The layout has been revised.

Abstract

Verification of cyber-physical systems (CPS) is a challenging task. A considerable effort has been invested to develop pragmatic methods, such as falsification testing, which facilitate generation of inputs that lead to the violation of the CPS requirements. The resulting counterexamples are used to locate and explain faults and debug the system. However, CPS rarely operate in fully unconstrained environments and not all counterexamples have the same value – a fault resulting from a common usage of the system has more impact than a fault that is triggered by an esoteric input sequence. This aspect is neglected by the existing falsification testing techniques. We propose a new falsification testing methodology that is aware of the system's expected usage. Given a user profile model in the form of a stochastic hybrid automaton, an executable black-box implementation of the CPS and its formalized requirements, we provide a test generation method that (1) uses efficient randomized methods to generate multiple violating traces, and (2) estimates the probability of each counterexample, thus providing their ranking to the engineer.

1 Introduction

Correctness is a crucial requirement during the design of safety-critical cyber-physical systems (CPS) such as smart homes, autonomous driving and intelligent medical devices. The interplay between computational (digital controllers, embedded software) and physical components (sensors and actuators), the increasing use of machine learning-based data-driven modules and the sophisticated interactions with unpredictable environments make the problem of correct and safe CPS design hard and challenging. Despite tremendous recent progress, formal verification does not scale yet to the size of realistic CPS applications. As a consequence, today's state-of-the-practice relies mainly on the more pragmatic simulation-based testing approaches.

The CPS community has invested in the recent past significant effort to improve the testing activities. Falsification-based testing (FBT) [1] is a popular method that renders the test generation process more systematic. FBT uses formal specifications with quantitative semantics to guide the system-under-test (SUT) to the violation of its requirements, whenever possible. It follows that FBT provides effective means to systematically detect a fault in the system. The resulting witness of the requirement violation is used to locate and explain the fault and hence to facilitate the system debugging task.

The classical FBT has a limitation – the test generation method focuses on finding one counterexample among possibly many of them. However, CPS often operate in partially constrained environments with certain assumptions on their usage in which not all counterexamples have the same value. For example, a fault triggered by a common usage of the system has much more

impact then another fault resulting from some rare and esoteric input sequence. This is an important aspect for prioritizing debugging tasks under time and budget constraints and that is completely neglected by the existing FBT techniques.

We introduce in this paper a new methodology for *usage-aware* FBT to remedy the above situation. In our approach, we assume that: (1) a user profile model that describes the system's intended usage is given in the form of a stochastic hybrid automaton, (2) the SUT is provided in the form of an executable black-box implementation, and (3) the requirements are formalized using temporal logic.

We first use a randomized accelerator procedure to generate test inputs from the user profile model. We then feed the input vector to the SUT and execute it. We use the temporal logic monitor to detect potential violation of requirements. Whenever we find a counterexample, we use statistical model checking (SMC) [2–4], and more specifically importance splitting (IS) [5], to estimate the likelihood of the counterexample. The estimated probability of counterexamples enables us to rank them according to their likelihood, thus facilitating prioritization of the debugging tasks.

We instantiated our usage-aware FBT methodology (described in Section 2) with concrete methods and tools and implemented it in a prototype framework. We adopted UPPAAL SMC [6] to model stochastic hybrid automata (Section 3), MATLAB Simulink [7] to implement black-box SUTs and signal temporal logic (STL) [8] to formalize CPS requirements (Section 4). We developed a modified variant of the randomized reachability analysis (RRA) [9] procedure as our randomized accelerator for efficiently finding counterexamples (Section 5) and adapted a version of the IS algorithm to estimate counterexample probabilities (Section 6), integrating both methods to the UPPAAL SMC engine. We used MATLAB Simulink's simulation environment to execute generated input sequences and the RTAMT [10] runtime verification library to monitor the resulting simulation traces against STL requirements. We used a thermal model of a house as our case study (Section 7) to evaluate our approach (Section 8).

Related Work

Falsification-based testing

Falsification-based testing (FBT) [1] is a test generation method that uses formal specifications equipped with quantitative semantics to guide the search for behaviors that violate the formalized requirements. In that work, the authors propose to use deterministic assumptions for restricting the test search space. The test search space can be additionally restricted using symbolic reachability methods [11]. The classical FBT approaches also stop the gener-

ation of tests after finding the first violation of a requirement. The adaptive FBT method [12] remedies this situation by introducing the notion of specification coverage and providing means to generate multiple qualitatively different counterexamples. None of these works allow one to compare violation witnesses according to their likelihood to happen. To contrast, we introduce probabilistic user profile models of the SUT to enable ranking countexamples.

Probabilistic Model Checking

Counterexamples play an important role in probabilistic model checking and have received considerable attention in the last two decades, see [13] for a survey on methods for generating probabilistic counterexamples. We mention the early work from Han and Katoen [14], who originally propose a method for finding the strongest evidence, i.e. the most likely counterexample violating an until-specification as a hop-constrained shortest path problem. The tool DiPro [15] allows generating probabilistic counterexamples discrete time Markov chains, continuous time Markov chains and Markov decision processes. In the work on probabilistic model checking, the model of the SUT is available as a white-box, which allows precise computation of counterexample probabilities but limits the scalability of the approach to the systems of small size and complexity. In our approach, we consider black-box SUTs of arbitrary size and complexity, and use simulation-based methods to detect counterexamples and estimate their probabilities.

Statistical and Randomized Testing

Statistical model checking (SMC) is a Monte Carlo simulation method used to estimate the probability of violating formal requirements. Reliable estimation of rare events remains difficult and is typically addressed by the *importance splitting* (IS) [16, 17]. IS divides the goal with small probability into a sequence of intermediate goals that are easier to reach. An alternative way to address the problem of rare-event simulation is to use *randomized reachability analysis* (RRA) [9]. RRA discards the stochastic semantics of the model to increase the chance of exercising a rare event. While RRA can efficiently find a counterexample, it cannot be used alone to estimate its probability. On the other hand, SMC and IS can reason about the probability that a given SUT violates a property, but are less appropriate to estimate the probability of a single counterexample. In our work, we use the synergies between SMC, IS and RRA to achieve efficient falsification while enabling the likelihood estimation of counterexamples.

2 Methodology

In this section, we describe our user-aware falsification-based testing methodology. The input to our approach are three artefacts: the *user profile* model, the black-box implementation of the *system-under-test (SUT)*, and a *formalized requirement*. The output of our approach is a list of input sequences (test cases) that lead to the falsification of the requirement ranked according to their estimated probability of happening.

User profile

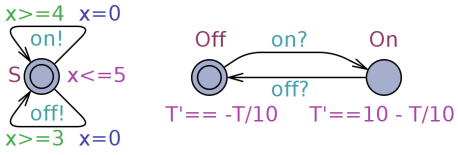
The user profile is a stochastic hybrid automaton that models the expected use of the SUT. It allows for rich and complex dynamics as well as stochastic behavior. Straightforward simulations of the user profile can be performed to generate inputs for the SUT. Generated simulations follow the underlying stochastic semantics of the model, which allows them to mimic the behavior of the real user and supports reasoning about the probability of generating a particular input sequence.

SUT

The SUT is a reactive dynamic system, which consumes an input sequence to generate another sequence of observable output quantities. We assume an executable black-box implementation of the SUT, whose behavior can be only observed at its input/output interface.

Formalized requirement

The formalized requirement is given in the form of a temporal logic specification. It defines the expected temporal and timing relations between input and output quantities and is used as an oracle to discriminate behaviors that satisfy the requirement from those that violate it.



$$\varphi = \Box_{[0,20]}(T \leq 75)$$

Fig. D.1: Running example. User profile model (top left), the heater controller (top right), and the formalized requirement (bottom).

Example 2.1 (Running example)
To illustrate the different steps of the methodology, we use a simplified heat controller, depicted in Figure D.1.

The simplified heat controller follows a deterministic implementation (Figure D.1 top right) has a single continuous state variable, the temperature T and consists of two discrete modes, `Off` and `On`. In the `Off` mode, the temperature decreases according to the differential equation $T' = -\frac{T}{10}$. Conversely, the temperature increases in the `On` mode according to the differential equation $T' = 10 - \frac{T}{10}$. The temperature range is limited to the interval between 0 and 100 degrees (not shown in the figure). The change between the two heater's modes is triggered by actions `on` and `off` provided by an external environment. We note that while the implementation is given in the form of a hybrid automaton for illustration purposes, we assume that it is seen as a black box to the tester, i.e. the tester can only provide the actions `off` and `on` and observe the temperature T. The stochastic user profile (Figure D.1 top left) models the expected generation of the `off` and `on` actions. The clock x measures the time between two consecutive actions. After an action happens, x is reset to 0 and a time delay between 0 and 5 is sampled according to the uniform distribution. If the time delay is in the interval $[0, 3)$ no action is enabled and an additional time delay must be taken. If it is in the interval $[3, 4)$, the action `off` is taken with probability 1. If the time delay is in the interval $[4, 5)$, the actions `off` and `on` are triggered, each with probability 0.5. It follows that the action `off` is likely to happen three times more often then the action `on`. Finally, the formalized requirement φ (Figure D.1 bottom) states that within 20 time units, the temperature T must continuously remain within 75 degrees.

2.1 Baseline Solution

We first propose a straightforward baseline solution to estimate the probability of an error in the system is by using SMC [18, 19]. The core idea of SMC is to generate simulations of a stochastic model and then statistically analyze them to estimate the probability of the system to violate requirements with some degree of confidence. Figure D.2 shows the workflow of the SMC application to our case – the input sequences are generated by the user profile and the outputs from SUT are analyzed by SMC to conclude on probability estimate of an error.

Discovery of the most stubborn bugs in the CPS often requires generation of "exotic" input sequences. In non-trivial stochastic models, generating an "exotic" input can often be considered an extremely rare event, i.e. its probability being in range $[0; 10^{-100}]$. Therefore, we anticipate methods like SMC,

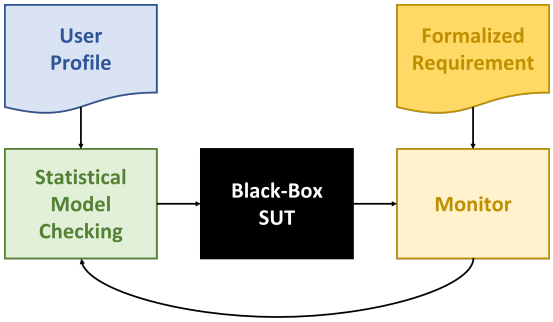


Fig. D.2: Workflow for falsification of black-box systems using SMC.

which exercises the most likely behavior of the system, to be impractical due to the time required to generate enough evidence (simulations) to achieve a reasonable statistical confidence.

In most real applications there is either a limit of resources for system verification or a requirement on the system being fail-proof up to a certain degree, as further bugs are more costly to fix than to replace a system. In both cases, it is crucial to focus resources on eliminating the most probable bugs first. While SMC estimates the overall existence of the bug, it cannot reason about a probability of any individual concrete input sequence. Hence, SMC does not help to conclude if a particular counterexample is of any concern in practice.

2.2 Efficient Solution

We recall that our proposed methodology not only (1) allows us to identify violations efficiently but also (2) estimates the probability for each discovered counterexample. It utilizes the Randomized Accelerator (RA) which is a modified version of the randomized reachability analysis algorithm, initially proposed by [9] as an efficient error detection method for timed and stopwatch automata models. Among the modifications, we extend the algorithm to support the rich dynamics of hybrid automaton models to allow simulation of the user profile. In contrast to SMC, RA discards the underlying stochastic semantics to favor exploration of otherwise unlikely to reach parts of the model. As a consequence, RA cannot reason about the probability of its generated simulations, but excels at finding "exotic" traces fast. In their study, [9] have shown their randomized reachability analysis to be up to three orders of magnitude faster than SMC at discovering bugs.

The workflow for a single iteration of our methodology is shown in Figure D.3. The process starts by applying RA on the user profile to generate input sequences for SUT. The latter is simulated with the given input and its

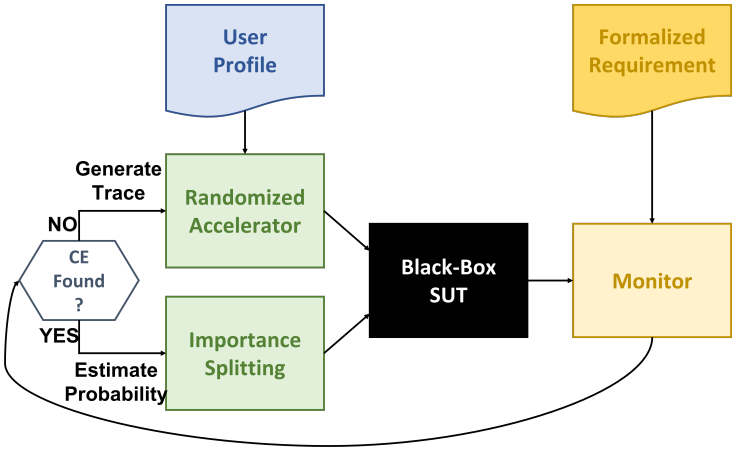


Fig. D.3: Workflow of the usage-aware falsification-based testing methodology.

output is monitored w.r.t. to the property of interest. The process is repeated until the counterexample input sequence that violates the property is discovered. Moreover, RA exploits the information from previous runs to favor "more promising" parts of the user profile that are deemed to have affected the monitored property towards being violated.

The counterexample contains the information about the execution of the user profile – transitions taken in the hybrid automaton and the outputted dynamics. To reason about the probability of counterexamples we use the Importance Splitting (IS) from [17]. IS allows one to estimate probability of rare events which SMC cannot do reliably or quickly. We use IS to estimate the probability of following the qualitative trace from the user profile, i.e. focusing on executing transitions of the user profile model in the sequence commanded by the trace. Additionally, a run of IS generates a number of traces with different timing behaviors w.r.t to the stochastic semantics. Since the timing behavior might have a crucial impact on the property satisfaction, we check all the traces generated by IS against SUT to estimate the ratio of traces violating the property.

The counterexample and its estimated probability becomes the result of a single iteration of our proposed methodology. Each counterexample is ranked according to its estimated probability to occur in practice. The search can then continue in a similar fashion to discover more bugs.

3 Hybrid Systems

In this section, we recall the definition of hybrid automata with stochastic semantics [20] that we use to model the user profiles. Let X be a finite set of

continuous variables. A variable valuation over X is a mapping $v : X \to \mathbb{R}$. We write $\mathbb{R}^X$ for the set of valuations over X. Valuations over X evolve over time according to a delay function $F : \mathbb{R}_{\geq 0} \times \mathbb{R}^X \to \mathbb{R}^X$, where for a delay d and valuation v, $F(d, v)$ provides the new valuation after a delay of d. As is the case for delays in timed automata, delay functions are assumed to be time additive in the sense that $F(d_1, F(d_2, v)) = F(d_1 + d_2, v)$. To allow for communication between different hybrid automata we assume a set of actions Σ, which is partitioned into disjoint sets of input and output actions, i.e. $\Sigma = \Sigma_i \uplus \Sigma_o$.

Definition 22
A Hybrid Automaton (HA) $\mathcal{H}$ is a tuple $\mathcal{H} = (L, l_0, X, \Sigma, E, F, I)$, where:

- L is a finite set of locations,
- l_0 is an initial location s.t. $l_0 \in L$,
- X is a finite set of continuous variables,
- Σ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_o) s.t. $\Sigma = \Sigma_i \uplus \Sigma_o$,
- E is a finite set of edges of the form (l, g, σ, r, l') where $l, l' \in L$, g is a predicate on R^X which acts as a *guard* that must be satisfied, $a \in \Sigma$ is an action label and u is a binary relation on $\mathbb{R}^X$ which acts as an *update*,
- $F(l)$ is a delay function for each location $l \in L$, and
- I assigns *invariant* predicates $I(l)$ to any location $l \in L$.

The semantics of a HA $\mathcal{H}$ is a timed labeled transition system, whose states are pairs $(l, v) \in L \times \mathbb{R}^X$ with $v \models I(l)$, and whose transitions are either delay transitions $(l, v) \xrightarrow{d} (l, v')$ with $d \in \mathbb{R}_{\geq 0}$ and $v' = F(d, v)$, or discrete transitions $(l, v) \xrightarrow{a} (l', v')$ if there is an edge (l, g, a, u, l') such that $v \models g$ and $u(v, v')$. We write $(l, v) \rightsquigarrow (l', v')$ if there is a finite sequence of delay and discrete transitions from (l, v) to (l', v'). We note that the effect of the delay function F may be specified by a set of ODEs that need to be solved and that governs the evolution of the continuous variables in time. The tool Uppaal SMC [21], which can perform simulations of HA, also supports built-in functions (such as *sin*, *cos*, *log*, *pow* and *sqrt*) that help to enrich the dynamics. When generating a run, Uppaal SMC does not solve ODEs exactly, but rather approximates the integration with the Runge-Kutta method.

We denote $\omega = s_0 d_1 a_1 s_1 d_2 a_2 \ldots$ to be a *timed word* where for all $i, s_i \in S$, $a_i \in \Sigma$, $s_i \xrightarrow{d_{i+1}} \xrightarrow{a_{i+1}} s_{i+1}$ and $d_i \in \mathbb{R}_{\geq 0}$. If ω is a finite timed word, we write $|\omega| = n$ to denote the length. We write $\omega[i]$ to denote a prefix run of ω up to i such that $w[i] = s d_1 a_1 s_1 \ldots d_i a_i s_i$. Last, we denote by $\Omega(\mathcal{H})$ the entire set of timed words over $\mathcal{H}$.

Example 3.1 (Running example)
Figure D.4 shows the resulting evolution of the continuous temperature variable from the SUT from Figure D.1 induced by the example input sequence (timed word)

$$3.7 \cdot \texttt{off} \cdot 4.1 \cdot \texttt{on} \cdot 3.1 \cdot \texttt{off} \cdot 4.5 \cdot \texttt{off} \cdot 4 \cdot \texttt{off}.$$

The property φ is satisfied as the temperature stays under 75 degrees for the 20 time units of the simulation.

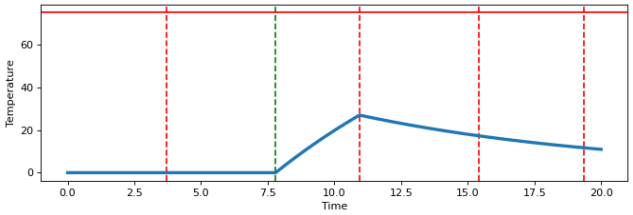


Fig. D.4: A timed word with on and off actions represented by green and red vertical dashes lines, respectively. The temperature dynamics of SUT (blue line) and φ threshold of 75 degrees (red line) is shown.

3.1 Stochastic Semantics of Hybrid Automata

Hybrid Automata may be given a stochastic semantics by refining the non-deterministic choices of transitions and delays by probabilistic and stochastic distributions. For each state $s = (l, v)$ of HA $\mathcal{H}$ there exists:

- the *delay density function* μ_s gives a probability distribution over delays in $\mathbb{R}_{\geq 0}$ that can be taken by a component, such that $\int \mu_s(t)dt = 1$,
- the *output probability function* γ_s gives a probability of taking an output $o \in \Sigma_o^j$ such that $\sum_o \gamma_s(0) = 1$, and
- the *next-state density function* η_s^a gives a probability distribution on the next state $s' = (l', v') \in \mathbb{R}^X$ given an action a such that $\int_{s'} \eta_s^a(s') = 1$.

For outputs happening deterministically at an exact time point d (or deterministic next states s'), $\mu_s(\eta_s^a)$ becomes a Dirac delta function $\delta_d(\delta_s')$ [1].

[1] which should formally be treated as the limit of a sequence of delay density functions with decreasing, non-zero support around d.

In UPPAAL SMC, the delay is always chosen according to uniform distribution in case of a bounded (by an invariant) delay or according to an exponential distribution if a delay can be indefinite. The choice of a transition is uniform, however UPPAAL SMC provides a syntax for *branching points* that allow specifying discrete probabilities for transitions. The probability distributions on the next-state can be defined with the help of the `random[b]` function that denotes an uniform distribution in continuous range $[0, b)$.

Consider $\mathcal{H}$ to be a stochastic HA. For $s \in S$ and $a_1 a_2 \dots a_k \in \Sigma^*$ we denote a *timed cylinder* $\pi(s, a_1 a_2 \dots a_k)$ to be the set of all timed words from s with a prefix $t_1 a_1 t_2 a_2 \dots t_k a_k$ for some $t_1, \dots, t_n \in \mathbb{R}_{\geq 0}$. An infinite timed word $\omega = s_0 d_1 a_1^\omega s_1 d_2 a_2^\omega \dots d_k a_k^\omega s_k \dots$ belongs to the timed cylinder, written as $\omega \in \pi(s, a_1, a_2, \dots, a_k)$, if $a_i^\omega = a_i$ for all i up to k and $s_0 = s$.

Example 3.2 (Running example)
Figure D.5 shows two timed words belonging to the same timed cylinder $\pi(s, \texttt{off}, \texttt{off}, \texttt{on}, \texttt{off}, \texttt{off})$, where $s = (\texttt{S}, \texttt{x=0})$ is the initial state.

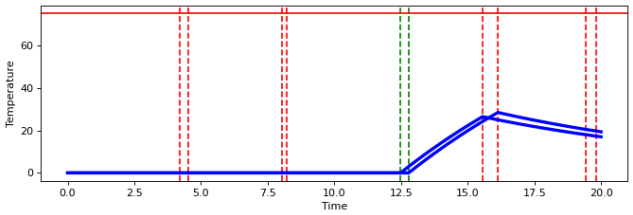


Fig. D.5: Two timed words belonging to the same cylinder.

Providing the basic elements of a Sigma-algebra we now recall from [20] the inductively defined measure for such timed cylinders:

$$\mathbb{P}_{\mathcal{H}}(\pi(s, a_1 a_2 \dots a_k)) = \int_{t \geq 0} \mu_s(t) \cdot \gamma_{s^t}(a_1) \cdot \int_{s'} \left(\eta_{s^t}^{a_1}(s') \cdot \mathbb{P}_{\mathcal{H}}(\pi(\mathbf{s'}, a_2 \dots a_k)) ds' \right) dt \tag{D.1}$$

The probability of following a timed cylinder π is computed by integrating over the initial delays t in the outermost level. Next, we take the probability of outputting a_i. The last part integrates over all successors s' and takes a product of probabilities for stochastic state changed after taking the delay t and outputting a_1 as well as the probability of following the remainder of the timed cylinder.

A general system can represented as a network of HA. Under the assumption of input-enabledness, an arbitrary number of HA can be composed into a network where the individual components communicate with each other and all together act as a single system. A race-based stochastic semantics determines which of the components in the network gets to perform an output such that the winning component is the one with the smallest chosen delay. Here we skip the definition of networks of HA and their stochastic semantics, and refer the interested reader to [20] for in-depth details.

4 Formalized Requirements

Let $\mathcal{F}(\omega_{\downarrow}) = y$ be a function representing a black-box nonlinear hybrid system that gives a real-valued output y on the given projection of a timed word $\omega \in \Omega(\mathcal{H})$ of a stochastic HA $\mathcal{H}$, denoted as $\omega_{\downarrow}$. Note that the projection of a timed word is obtained with the help of the Runge-Kutta method used to approximate the integration of the user profile ODEs with a fixed time step δt defined by the user. Effectively, the projection represents a simulation of the model $\mathcal{H}$ that expresses rich dynamics w.r.t. to the stochastic semantics.

Next define a property φ expressed in a Signal Temporal Logic (STL) language [8] which we now recap. An STL formula φ consists of atomic predicates together with Boolean and temporal operators. The temporal operators include *always* ($\Box$), *eventually* ($\Diamond$) and *until* ($\mathcal{U}$) and are restricted to intervals of form $[a, b]$, where $0 \leq a < b$ and $a, b \in \mathbb{R}_{\geq 0}$. Now let $\sim \in \{\leq, <, >, \geq, =, \neq\}$ be the set of relational operators. The atomic predicates are defined over a scalar-valued function $f(y(t)) \sim c$ evaluated over an input y at time t and where $n \in \mathbb{N}$. The grammar of STL language is then given as:

$$\varphi := \mathrm{T} \mid f(y(t)) \sim c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_I \varphi_2 \tag{D.2}$$

where I is a non-empty interval defined over extended reals.

The temporal operators can be defined in terms of Boolean and logic operators such that $\Diamond_{[a,b]}\varphi \equiv \mathrm{T}\,\mathcal{U}_{[a,b]}\varphi$ and $\Box_{[a,b]}\varphi = \neg\Diamond_{[a,b]}\neg\varphi$. If the interval is omitted, it is assumed to be $[0, \infty)$. Furthermore, the quantitative semantics is defined for STL by [22] with the help of function $\rho(\varphi, y, t)$ which gives *robustness*, i.e. degree of satisfaction of the formula φ by (y, t). The formula is satisfied if the robustness is positive and the other way around. Given a *robustness* function ρ, when $\rho(\varphi, y)$ is positive it indicates that y satisfies φ, written as $y \models \varphi$. In this paper, we restrict our attention to the *bounded* fragment of STL

Let **Error** $\in S$ be a set of error states of HA $\mathcal{H}$. An infinite timed run $\omega = s_0 \xrightarrow{d_1} \xrightarrow{a_1} s_1 \xrightarrow{d_2} \xrightarrow{a_2} \ldots$ is an *error run* if $\exists i. \exists \tau \leq d_i. s_{i-1}^{\tau} \in$ **Error**, where $\tau \in \mathbb{R}_{\geq 0}$ and s_{i-1}^{τ} is the state such that $s_{i-1} \xrightarrow{\tau} s_{i-1}^{\tau}$. We say that ω has an

error at i'th transition and show it as $\mathcal{F}(\omega[i]_{\downarrow}) \not\models \varphi$. In this case, clearly any ω' such that $\omega' = \omega[i]\omega_n$ is also an error run. We now proceed to explain the next step – generation of violating traces.

5 Falsification Testing w/ Randomized Accelerator

In this section, we describe our FBT approach for discovery of traces that violate requirements. In order to efficiently find counterexamples, we use

the modified version of the randomized reachability analysis (RRA) initially proposed by [9] as a lightweight, quick and efficient error detection technique for timed systems. The core idea of RRA is to discard the underlying stochastic semantics of the model in an attempt to exercise an "exotic" behavior faster than with UPPAAL SMC. The method is based on exploring the model by means of repeated *random walks* that operate on concrete states and delays, and avoid expensive computations of symbolic, zone-based abstractions. Algorithm 4 shown an adapted version of RRA for simulation purposes. The random walks are issued until a state satisfying the property ψ_t is discovered. In the simulation setting, the property ψ_t is a simulation termination condition (e.g. time bound, step bound, etc.) and therefore the algorithm always terminates. In our case ψ_t is an integer constraint that requires the global (observing) clock to have elapsed until a specified value.

Algorithm 4 Randomized Reachability Simulation

```
1:  procedure RRA SIMULATE(s_0, ψ_t)
2:      while within time budget do
3:          s ← RANDOMWALK(s_0)
4:          if s ⊨ ψ_t then
5:              return ω = s_0, d_1 a_1 s_1, ..., d_n a_n s_n
6:  procedure RANDOMWALK(s)
7:      while s ⊭ ψ_t do
8:          a ← SELECTOUTPUT(s)
9:          d ← SELECTDELAY(a)
10:         s ← s' such that s --d--> --a--> s'
11:     return s
```

Once the simulation is completed, a finite timed word $\omega[n]$, that consists of the starting state and all delays and actions taken along the way, is returned (line 5).

Different heuristics can be used for `SelectOutput` and `SelectDelay` functions in the random walks (line 6). Among a number of heuristics presented by [9], we use the *random enabled transition* (RET) heuristic. RET chooses an *eventually enabled* transition uniformly at random, i.e. a transition that is

either currently enabled or can become such after a delay. More formally, consider $\mathcal{H}$ to be HA. Let $TB : S \times \Sigma \to \mathcal{P}(\mathbb{R}_{\geq 0})$ give the lower and the upper bound of the transition's availability range over the actions of a given HA. Simply put, TB gives both the smallest and the largest delay after which a certain action can be taken. Formally:

$$TB(s) = \{\underset{d \in \mathbb{R}_{\geq 0}}{\textbf{arg min}}\ s \xrightarrow{d} \xrightarrow{a} s'\} \cup \{\underset{d \in \mathbb{R}_{\geq 0}}{\textbf{arg max}}\ s \xrightarrow{d} \xrightarrow{a} s'\} \tag{D.3}$$

RRA simulation generates timed words $w \in \Sigma$ of HA $\mathcal{H}$ s.t.:

$$w = s_0 \xrightarrow{d_1} \xrightarrow{a_1} s_1 \xrightarrow{d_2} \xrightarrow{a_2} s_2 \xrightarrow{d_3} \xrightarrow{a_3} \dots \xrightarrow{d_n} \xrightarrow{a_n} s_n \tag{D.4}$$

where $s_n \models \psi_t$, and a_i is drawn uniformly at random from the set $\{a \mid s_{i-1} \xrightarrow{d} \xrightarrow{a} .d \in \mathbb{R}_{\geq 0}\}$ for all i, and d is also drawn uniformly at random from $TB(s_j, a_{j+1})$ for all j.

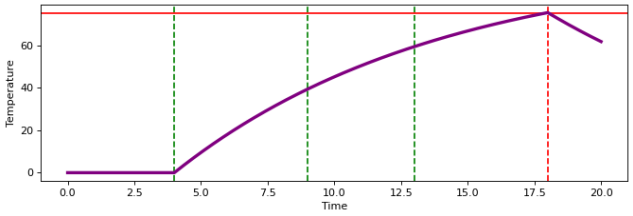


Fig. D.6: A violating timed word (counterexample).

Example 5.1 (Running example)
For our running example, Figure D.6 shows a RRA generated timed word that violates the requirement φ.

In this study we use RRA as our RA (randomized accelerator from Section 2) as a mean of accelerating discovery of rare events in the user profile model. In the remainder of this section, we discuss improvements over the vanilla version of the RRA algorithm.

5.1 Guiding of RRA

The RA from our methodology is applied for a number of iterations until a violating trace (counterexample) ω is discovered, i.e. until the robustness of the formula becomes negative $\rho(\varphi, \omega_{\downarrow}) < 0$. Clearly, in practice not all of the RA generated traces will violate the property, but some of them are likely

to be close. Since evaluation of the generated input against SUT is expensive, we are interested in minimizing the number of RA iterations. Thus, we guide our RA towards the areas of the state space of HA which are deemed to be "promising" according to the previous runs. More specifically, for non-violating runs we analyze the robustness of the output and search for robustness that lays outside its standard deviation and towards a violation. The corresponding actions of HA are then identified and prioritized over other actions in the following RA iteration. To avoid broadcasts that lead to a local minima, the "promising" actions are used only for a single iteration and are discarded if the property is not violated. Afterwards, an unguided run of RA is carried out and new promising transitions are recorded for the next RA iteration in the similar fashion. Thus, only every even run of RA is guided by promising actions.

5.2 Adaptive Simulation Duration

Short counterexamples are not only more probable to occur in practice but also easier to debug. For those reasons, we employ an adaptive simulation duration (ASD) that may change with each discovered counterexample. First, a user profile is simulated for a given duration. When a counterexample is found, we detect the time of first violation. In the next iteration the simulation duration is limited to that time. Intuitively, this is similar to the search for the shortest trace. However, at some point we may end up reducing the simulation time to a point where no counterexamples exist. Hence, we increase the current simulation time by 10% after a certain number of iterations is spent without finding a violation. We define that number of iterations to be the average number of iterations among so far discovered counterexamples plus a constant to ensure "enough" effort is spent before increasing the simulation duration.

6 Estimating Counterexample Probability

The probability of a timed cylinder is given by Equation D.1. Unfortunately, it is a theoretical construct that we cannot compute efficiently in practice. Alternatively, we can use simulation-based methods, such as SMC, to estimate the probability of following a timed cylinder. However, for rare events, SMC cannot estimate such probability reliably as simply too many simulations would be required to achieve a reasonable confidence. As a solution to this problem we use a rare event simulating technique known as *importance splitting* (IS).

Algorithm 5 Importance Splitting algorithm

```
1:  procedure IMPORTANCE SPLIT(π(s, a_1, a_2, ..., a_n))
2:      Successors_0 = {s}
3:      for i ← 1 ... n do
4:          Successors_i ← ∅
5:          for j ← 1 ... m do
6:              s ∈ Successors_{i−1}                    ▷ uniformly at random
7:              Let s −d→−a→ s'                         ▷ w.r.t. to the stochastic semantics
8:              if a = a_i then
9:                  Successors_i ← Successors_i ∪ {s'}
10:         p_i ← |Successors_i| / m
11:     return ∏_{i=1}^{n} p_i
```

6.1 Importance Splitting

The idea of IS is to split the final reachability goal δ_f into a number of intermediate sub-goals $\delta_1, \delta_2, \ldots, \delta_n$ that eventually lead to the main goal, i.e. $\delta_n = \delta_f$. The sub-goals are called *levels* that each get closer to the goal which naturally can be ensured by a *score function*. The score of each subsequent goal is required to be larger than that of a previous one. In our case a score function is binary function that helps to ensure that the timed cylinder is followed, i.e. the action transitions are taken strictly in a sequence defined by π.

The procedure of our IS is shown in Algorithm 5 which is an adapted version of Algorithm 2 from [17] with the *fixed effort* scheme that we explain later. For each level n we perform a fixed number m of simulations. A state is chosen from the set of previous $\texttt{Successors}_{i-1}$(line 6) uniformly at random and a successor is generated randomly according to the stochastic semantics (line 7). The generated successor is recorded in the set of $\texttt{Successors}_i$, but only if it was obtained by following a corresponding action a_i from the timed cylinder. The probability estimate of reaching the level i from level $i-1$ is then $\frac{\texttt{Successors}_i}{m}$. Repeating this process for all levels n produces probability estimate of a timed cylinder of a HA $\mathcal{H}$ defined as follows:

$$\mathbb{P}_{\mathcal{H}}(\pi(s, a_1 a_2 \ldots a_n)) \simeq \prod_{i=1}^{n} \frac{|\texttt{Successors}_i|}{m} \tag{D.5}$$

6.2 Counterexample Probability

With the help of IS, we can now estimate the probability of the timed cylinder $\mathbb{P}(\pi(s, a_1, a_2, \ldots, a_n))$ which was generated by our RA. To ease the notation

we sometimes write (π_n) instead of $\pi(s, a_1, a_2, \ldots, a_n)$. A run of IS also produces a number of concrete simulations that all follow the timed cylinder and are generated according to the stochastic semantics of the underlying HA. During a run of IS, each successor has its predecessor recorded together with the delay and broadcast action that lead to that successor. The concrete simulations are obtained by back-tracing from the successors that made it to the very last level and back to the starting state. We define a finite set of timed words (traces) w generated by IS from a timed cylinder π_n as $\Gamma_{\pi_n} \subseteq \pi_n$ such that $|\Gamma_{\pi_n}| \leq m$, where m is the effort allocated per level of IS fixed effort scheme.

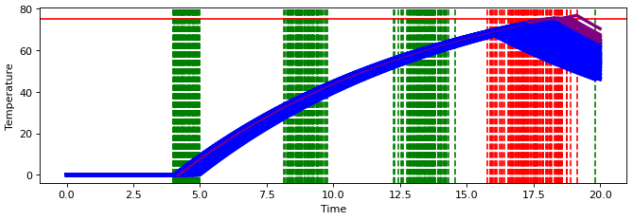


Fig. D.7: One hundred traces generated by IS. Blue line traces satisfy the property and purple line ones violate.

Even though all traces in Γ_{π_n} follow the same timed cylinder, the differences in the timing behavior (chosen delays) may influence the dynamics of the user profile to a large degree. Violation (or satisfaction) of the property monitored for SUT - as well as the level at which a violation appear - is therefore not guaranteed to be identical for all traces Γ_{π_n} reported by IS.

We first estimate the probability of violating the STL property with a timed cylinder π_n, i.e. $\mathbb{P}(\omega \in \pi_n \wedge \mathcal{F}(\omega_\downarrow) \not\models \varphi)$. In the following let m be a number of IS simulation per level and let $\texttt{Successors}_i$ be the successors of each level i in IS. Considering the IS sampled traces Γ_{π_n} and their (varying) violation of the monitored property, we may estimate this probability as follows:

$$\begin{aligned}
&\mathbb{P}(\omega \in \pi_n \wedge \mathcal{F}(\omega_\downarrow) \not\models \varphi) = \\
&\qquad \mathbb{P}(\pi_n) \cdot \mathbb{P}(\omega \in \pi_n \wedge \mathcal{F}(\omega_\downarrow) \not\models \varphi \mid \pi_n) \simeq \\
&\qquad \prod_{i=1}^{n} \left(\frac{|\texttt{Successors}_i|}{m} \right) \cdot \left(\sum_{\omega \in \Gamma_{\pi_n}} \mathcal{F}(\omega_\downarrow) \not\models \varphi \right) \cdot \frac{1}{|\Gamma_{\pi_n}|}
\end{aligned} \tag{D.6}$$

However, the above equation does not take into account that for a given trace ω, the property can be violated earlier than at the very last step n, i.e. $\mathcal{F}(\omega'_\downarrow) \not\models \varphi$, where $\omega' \subset \omega$ and $|\omega'| < |\omega|$. Furthermore, performing IS on the timed cylinder of length n may lose information about timed sub-cylinders of shorter length k, $|\pi_k| < |\pi_n|$. We will write $\mathcal{F}_k(\omega_\downarrow) \not\models \varphi$ if k is

the minimal index such that $\mathcal{F}(\omega[k]_{\downarrow}) \not\models \varphi$. Also let Θ be the function that given a finite set of traces Γ, a property φ and an index i returns only prefix words $\omega[i]$ such that the first violation occurs at step i, formally defined as:

$$\Theta(\Gamma, \varphi, i) = \{\omega[i] \mid \omega \in \Gamma \wedge \mathcal{F}_i(\omega_{\downarrow}) \not\models \varphi)\} \tag{D.7}$$

Now taking the level of occurrence of property violation into account leads to a higher error probability that may be estimated as follows:

$$\begin{aligned} &\mathbb{P}(\omega \in \pi_k \wedge \mathcal{F}_k(\omega_{\downarrow}) \not\models \varphi \wedge k \leq n) = \\ &\quad \sum_{k=1}^{n} \left(\mathbb{P}(\pi_k) \cdot \mathbb{P}(\omega \in \pi_k \wedge \mathcal{F}_k(\omega_{\downarrow}) \not\models \varphi \mid \pi_k) \right) \simeq \\ &\quad \sum_{k=1}^{n} \left(\left(\prod_{i=1}^{k} \frac{|\texttt{Successors}_i|}{m} \right) \cdot \frac{|\Theta(\Gamma_{\pi_k}, \varphi, k)|}{|\texttt{Successors}_k|} \right) \end{aligned} \tag{D.8}$$

The key idea of this approach is to separate a timed cylinder of length n into sub-cylinders of length k such that $|\pi_k| \leq |\pi_n|$. Summing up the probabilities of sub-cylinders π_k and violation occurring at the very last step k gives an upper bound of the probability estimate for the timed cylinder to violate the property. However, this formula requires not only to compute the set of traces Γ_{π_k} for each k, but to also check each of the traces $\omega \in \Gamma_{\pi_k}$ against the Simulink model. With latter of the steps being the most computationally demanding in our proposed methodology, we believe that equation D.8 will be too expensive in practice. Instead, we use another lower bound probability estimate

$$\sum_{k=1}^{n} \left(\left(\prod_{i=1}^{k} \frac{|\texttt{Successors}_i|}{m} \right) \cdot \frac{|\Theta(\Gamma_{\pi_n}, k, \varphi)|}{|\texttt{Successors}_k|} \right) \tag{D.9}$$

which requires computing only the traces Γ_{π_n} for the main cylinder. This approach gives a smaller, lower bound probability estimate, but it is much cheaper to compute in practice.

Example 6.1 (Running example)
For our running example, we consider 100 traces generated by IS (Figure D.7) for a timed cylinder with an action sequence `on`, `on`, `on`, `off`, `on`. Violation of 75 degrees threshold is observed in 9 (out of 100), all at step 4. The probability estimate of Equation D.9 is then $\left(\frac{23}{100} \cdot \frac{27}{100} \cdot \frac{25}{100} \cdot \frac{77}{100}\right) \cdot \frac{9}{100} \approx 0.001$, where the parenthesis give IS probability estimate of the cylinder with 4 steps.

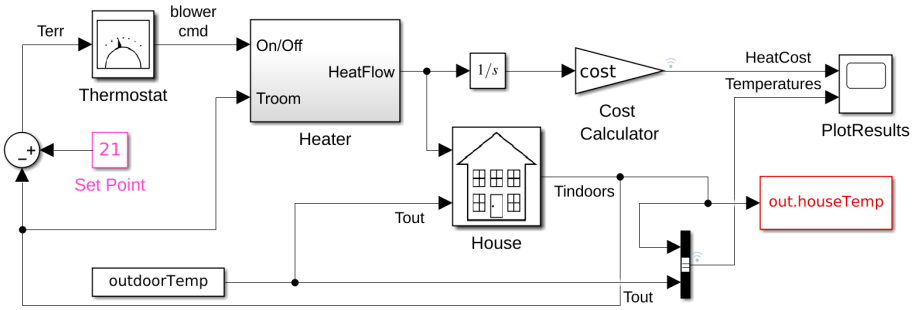


Fig. D.8: Simulink Thermal Model of a House.

7 Case Study

We use a Thermal Model of a House[2] as our black-box SUT. The model, shown in Figure D.8 accounts for the heating system, thermal dynamics of the house, the outdoor environment and a number of associated properties such as house geometry, thermal resistance of the materials, heater flow rate and hot air temperature, etc. The heating of the house is controlled by the thermostat that turns on/off the heater once the temperature is below/above specified thresholds. The inside temperature is calculated by the model by considering the heat flow from the heater and the heat losses to the environment through the insulation of the house.

User Profile

To simulate the outdoor environment for the thermal house system, we use the HA weather model which provides "realistic" complex dynamics of the potential temperature. The model is shown in Figure D.9. The daily and yearly temperature fluctuations are modelled by sinusoidal waves with varying phase, amplitude and biases. In our experiments we simulate the weather model for a period of 1 year while the starting period of the simulation being January 1st; therefore, the waves are adjusted accordingly and depend on an elapsing, observing clock x that we use to model the time in hours. In addition to daily and yearly temperature changes the model supports small and large "anomalous" temperature fluctuations that are enforced to occur at least every so often by `guards` and `invariants` on edges and locations, respectively. Each fluctuation results in a temperature change of the magnitude and dynamics determined by ODEs in the locations. A number of large fluctuations can happen sequentially, representing e.g. a heat wave or a sudden temperature drop. The type of a fluctuation, as well as its duration, which

[2] `https://se.mathworks.com/help/simulink/slref/thermal-model-of-a-house.html`

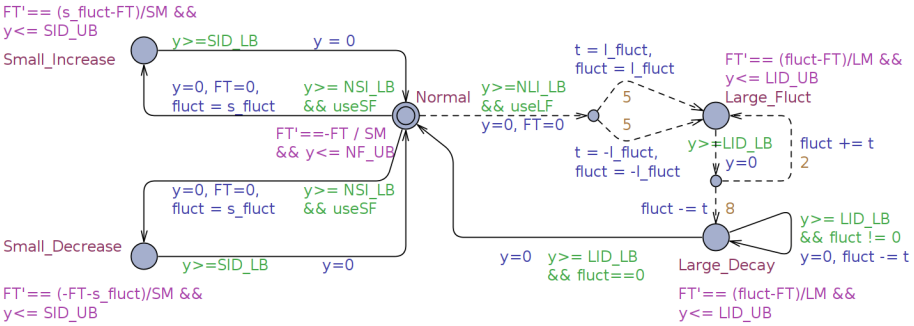


Fig. D.9: Weather profile hybrid automaton model.

largely affects the dynamics, is decided during the simulation and in accordance with the stochastic semantics described in Section 3. The likelihood of an additional large fluctuation taking place right after a previous one is defined by discrete probabilities 8 and 2 on the outgoing transition from the `Large_Fluct` location. Even though no restrictions are made on the maximal number of large fluctuations happening sequentially, the probability of additional n sequential large fluctuations (after the initial one) is $(\frac{2}{8+2})^n = 0.2^n$.

Property Monitor

Finally, as a property monitor we use python library RTAMT [10] which supports both offline and online monitoring of STL properties. The inside temperature of the house is monitored in an offline setting to measure the property robustness. We monitor the property $\square(T \leq 16 \rightarrow \Diamond_{[0,24]}(T \geq 18))$, i.e. it is required that the temperature, if it drops below 16 degrees, always recovers to at least 18 degrees within 24 hours.

8 Experiments

8.1 Baseline Solution

We use SMC as a baseline and compare it to the performance of our falsification methodology. The two approaches are rather different as SMC cannot reason about probability of each individual violating trace. Rather, SMC estimates an overall property violation probability which lays within some approximation interval $p \pm \epsilon$ with a confidence $1 - \alpha$. The amount of simulations N required to produce an approximation interval given ϵ and α can be computed using Chernoff–Hoeffding inequality as follows:

$$N \geq \frac{\log(2/\alpha)}{2\epsilon^2} \tag{D.10}$$

To accurately estimate a very improbable error in the system, the probability uncertainty ϵ must be sufficiently small. As can be seen in Table D.1, the growth of the required simulations is logarithmic and exponential in relation to α and ϵ, respectively. However, in practice this approach is too conservative. As an alternative, Uppaal SMC uses a sequential approach of Clopper-Pearson that computes the approximation interval with each iteration (for given α) and until the target ϵ is reached. Moreover, the further away a true probability is from $\frac{1}{2}$, the fewer simulations are needed. Empirical evidence[3] suggests that a true probability in range $[0, 10^{-5}]$ in practice requires roughly around 10% simulations (depending on α) of what inequality D.10 suggests. Nonetheless, for $\alpha = 0.01$ and $\epsilon = 5 \times 10^{-4}$ around as many as 10^6 simulations would be required.

Table D.1: Number of simulations required to produce an approximation interval $[p - \epsilon, p + \epsilon]$ with confidence $1 - \alpha$ using Chernoff–Hoeffding inequality.

Confidence α	Probability uncertainty ϵ			
	0.05	5×10^{-3}	5×10^{-4}	5×10^{-5}
0.1	600	59,915	5,991,465	599,146,455
0.05	738	73,778	7,377,759	737,775,891
0.01	1,060	105,967	10,596,635	1,059,663,474

In our experiments we only give an estimate of the time required by SMC to derive approximation intervals for a sufficiently small ϵ and α as each simulation is costly due to the execution of the Simulink model. To estimate the time of one iteration of the workflow from Figure D.2 we ran 20,000 SMC simulations with 220 of them violating the property. With the total time of 17h 36m 20s, a single iteration in average takes 3.169 seconds. Figure D.10 gives black-box SUT dynamics of one such iteration that includes outdoor

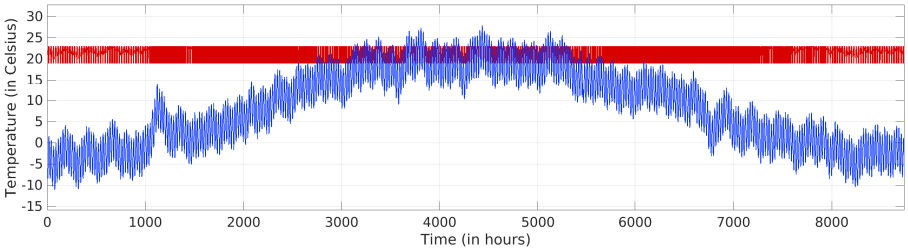


Fig. D.10: Simulink Thermal House indoor (red) and outdoor (blue) temperatures simulated for 1 year (8736 hours).

[3] `https://docs.uppaal.org/language-reference/requirements-specification/ci_estimation/`

ambient temperature (input) and a resulting indoor temperature (output) obtained after execution of SUT. As Uppaal SMC primarily exercises common behavior of the weather profile, we observe no requirement violation in the output from SUT.

Table D.2: IS fixed effort scheme sensitivity to the estimated probability variance w.r.t. number of simulation m per level. Each row represents 100 IS runs and a single exact probability calculation. Time is given in HH:MM:SS.

IS pr. stdev	IS pr. mean	Exact pr.	$\frac{\text{IS pr. mean}}{\text{Exact pr.}}$	Simulations per level	# levels (# trans.)	$\frac{\text{Fails}}{\text{Successes}}$	Total time
4.4554e-16	1.9357e-16	1.2628e-16	+53.3%	50	42	21/79	00:03:41
2.6050e-16	1.4082e-16	1.2628e-16	+11.5%	100	42	2/98	00:08:00
7.6806e-17	9.9405e-17	1.2628e-16	-21.3%	200	42	0/100	00:16:21
6.6849e-17	1.2627e-16	1.2628e-16	+0.0%	500	42	0/100	00:41:16
5.3405e-17	1.2639e-16	1.2628e-16	+0.1%	1000	42	0/100	01:23:37
2.1045e-26	8.3141e-27	4.1159e-27	102.0%	50	74	46/54	00:05:04
9.6553e-27	4.2122e-27	4.1159e-27	2.3%	100	74	4/96	00:15:17
3.3171e-27	3.2673e-27	4.1159e-27	-20.6%	200	74	0/100	00:31:18
3.4904e-27	4.1488e-27	4.1159e-27	0.8%	500	74	0/100	01:18:32
2.1917e-27	4.1945e-27	4.1159e-27	1.9%	1000	74	0/100	02:39:26
2.4205e-41	5.4142e-42	4.4194e-42	22.5%	50	118	25/75	00:09:59
4.8414e-42	2.7542e-42	4.4194e-42	-37.7%	100	118	1/99	00:24:42
3.7592e-42	2.9829e-42	4.4194e-42	-32.5%	200	118	0/100	00:49:17
4.0601e-42	4.1128e-42	4.4194e-42	-6.9%	500	118	0/100	02:07:44
2.9377e-42	5.0765e-42	4.4194e-42	14.9%	1000	118	0/100	04:16:32

8.2 IS Simulation Sensitivity

In addition to IS that is used to estimate the probability to follow a timed cylinder, we implement an exact method for probability computation. This method supports only a subset of HA models where all clocks are reset in every transition (this is the case for our weather profile). It enables us to compare how far the estimated probability is away from the true one.

We perform an experiment to determine to which degree the variance in the probability estimates produced by IS is affected by the number m of IS simulations performed per level by IS Algorithm 5. We vary the number of simulations per level and the total amount of levels, reporting the results in Table D.2. As anticipated, in cases with a small number of 50 simulation per level, a considerable amount of IS attempts (up to 50%) have failed to follow an entire timed cylinder ($\frac{Fails}{Successes}$ column). That is due to the IS algorithm being unable ("unlucky") to get through one of the "difficult" levels with only 50 attempts per level. However, to our surprise we have not been able to see a clear pattern indicating the amount of IS simulations per level to significantly affect the variance of the probability estimates. To confirm this, we performed several additional experiments both with the same parameters

Table D.3: Falsification of the case study model following the workflow from Section 2. Given are 20 counterexamples ranked according to the "Complex" probability from Equation D.9. Time is given in HH:MM:SS.

Discovery order	Exact cylinder pr.	IS pr. mean	Trace ratio (TR)	IS pr. · TR (Equation D.6)	Complex pr. (Equation D.9)	# transitions	# of RA iterations	Execution time
11	1.7778e-03	2.2464e-03	49/80	1.3759e-03	1.3759e-03	4	5	00:01:53
15	7.9012e-04	7.3568e-04	45/80	4.1382e-04	4.1382e-04	6	24	00:02:12
7	3.5556e-04	3.5211e-04	100/100	3.5211e-04	2.9221e-04	6	1	00:01:06
9	8.8889e-05	1.5640e-04	25/25	1.5640e-04	2.7784e-04	5	4	00:00:32
8	1.4222e-05	2.1236e-05	72/72	2.1236e-05	2.6238e-04	7	8	00:00:30
13	1.9753e-04	3.1786e-04	9/16	1.7879e-04	1.7879e-04	6	27	00:00:52
10	4.4444e-04	2.5920e-04	12/18	1.7280e-04	1.7280e-04	4	2	00:00:26
5	1.4047e-05	1.3293e-05	41/41	1.3293e-05	1.1223e-04	14	20	00:00:41
6	1.5803e-04	8.8559e-05	84/84	8.8559e-05	1.0326e-04	7	1	00:01:15
17	7.0233e-05	1.1707e-04	100/100	1.1707e-04	9.0020e-05	10	6	00:01:16
12	1.9753e-04	1.7952e-04	10/20	8.9760e-05	8.9760e-05	6	53	00:01:20
20	3.5117e-06	8.5251e-06	18/18	8.5251e-06	5.2098e-05	10	79	00:01:23
19	8.7791e-05	4.6627e-05	18/27	3.1085e-05	3.1085e-05	8	46	00:01:21
16	1.4047e-05	1.5274e-05	87/87	1.5274e-05	2.3174e-05	10	4	00:00:32
18	7.8037e-06	3.5226e-06	20/20	3.5226e-06	5.9297e-06	11	1	00:00:29
14	1.9753e-05	9.6188e-06	11/19	5.5687e-06	5.5687e-06	7	70	00:01:40
4	2.1923e-08	2.2023e-08	100/100	2.2023e-08	3.7905e-07	26	3	00:00:21
3	1.9560e-40	1.3114e-41	100/100	1.3114e-41	2.4558e-14	111	1	00:00:20
2	7.2106e-52	1.3739e-53	100/100	1.3739e-53	4.5403e-51	139	1	00:00:30
1	4.0514e-55	1.9894e-55	71/100	1.4125e-55	1.8720e-55	155	1	00:02:13
Total:	4.2413e-03	4.5709e-03	-	3.0438e-03	3.4873e-03	-	357	**00:21:01**
An estimate of time required by SMC (with $\alpha = 0.01$, $\epsilon = 5 \times 10^{-3}$) for 10000 iterations (based on Table D.1).							$10000 \cdot 3.169$ seconds =	**08:48:00**

and with different ones, but the observation remained the same. In the light of this and the fact that the execution time taken is roughly proportional to the number of IS simulations per level, in further experiments we fix the number of IS simulation per level to 100.

8.3 Efficient Falsification Evaluation

We evaluate our FBT methodology on the proposed case study and report the results in Table D.3. Due to ASD, the quality of discovered errors tends to increase over time as the length (# transitions) of the counterexample decreases. The probability estimate of Equation D.9 (E9) tends to be smaller for longer traces than that of Equation D.6 (E6); however, for short traces E9 and E6 are equal as the error occurs at the very last transition of the trace. Discovery, evaluation and ranking of 20 bugs with our falsification methodology is an order of magnitude faster than an estimated performance of SMC to conclude on the overall likelihood of a bug in SUT.

9 Conclusion and Future Work

We introduced in this paper a new methodology for usage-aware FBT of CPS. It combines stochastic HA modeling, randomized reachability analysis, statistical model checking, importance splitting and runtime verification to efficiently generate input sequences that lead to the violation of the requirements, while estimating their probability of happening in the real usage of

the system. We believe that the proposed methodology can significantly help the debugging effort by enabling to prioritize bugs with higher impact.

As future work, we plan to 1) develop more sophisticated guiding for our randomized accelerator, 2) introduce a state coverage metric, and 3) explore symbolic reachability techniques for HA.

References

[1] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas, "Monte-carlo techniques for falsification of temporal properties of non-linear hybrid 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. 211–220.

[2] H. L. S. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling," in *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, 2002, pp. 223–235.

[3] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, 2004, pp. 202–215.

[4] E. M. Clarke and P. Zuliani, "Statistical model checking for cyber-physical systems," in *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, 2011, pp. 1–12.

[5] C. Jégourel, A. Legay, and S. Sedwards, "Importance splitting for statistical model checking rare properties," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, 2013, pp. 576–591.

[6] 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.

[7] D. K. Chaturvedi, *Modeling and simulation of systems using MATLAB® and Simulink®*. CRC press, 2017.

[8] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and*

Fault-Tolerant Systems, Y. Lakhnech and S. Yovine, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 152–166.

[9] A. Kiviriga, K. G. Larsen, and U. Nyman, "Randomized Reachability Analysis in Uppaal: Fast Error Detection in Timed Systems," in *FMICS 2021*, A. Lluch Lafuente and A. Mavridou, Eds. Springer, 2021, pp. 149–166.

[10] D. Ničković and T. Yamaguchi, "Rtamt: Online robustness monitors from stl," in *Automated Technology for Verification and Analysis*, D. V. Hung and O. Sokolsky, Eds. Cham: Springer International Publishing, 2020, pp. 564–571.

[11] S. Bogomolov, G. Frehse, A. Gurung, D. Li, G. Martius, and R. Ray, "Falsification of hybrid systems using symbolic reachability and trajectory splicing," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, 2019, pp. 1–10.

[12] E. Bartocci, R. Bloem, B. Maderbacher, N. Manjunath, and D. Nickovic, "Adaptive testing for specification coverage in CPS models," in *7th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2021, Brussels, Belgium, July 7-9, 2021*, 2021, pp. 229–234.

[13] E. Ábrahám, B. Becker, C. Dehnert, N. Jansen, J. Katoen, and R. Wimmer, "Counterexample generation for discrete-time markov models: An introductory survey," in *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, 2014, pp. 65–121.

[14] T. Han and J. Katoen, "Counterexamples in probabilistic model checking," in *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, 2007, pp. 72–86.

[15] H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov, "Dipro - A tool for probabilistic counterexample generation," in *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, 2011, pp. 183–187.

[16] G. Rubino and B. Tuffin, *Rare event simulation using Monte Carlo methods*. John Wiley & Sons, 2009.

[17] K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, "Importance splitting in uppaal," in *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part III*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 13703. Springer, 2022, pp. 433–447. [Online]. Available: https://doi.org/10.1007/978-3-031-19759-8_26

[18] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 202–215.

[19] H. L. S. Younes, "Verification and planning for stochastic processes with asynchronous events," Ph.D. dissertation, 2004.

[20] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikuč ionis, D. B. Poulsen, and S. Sedwards, "Statistical model checking for stochastic hybrid systems," *Electronic Proceedings in Theoretical Computer Science*, vol. 92, pp. 122–136, aug 2012.

[21] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal smc tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, Aug 2015. [Online]. Available: https://doi.org/10.1007/s10009-014-0361-y

[22] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *Formal Modeling and Analysis of Timed Systems*, K. Chatterjee and T. A. Henzinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–106.

ISSN (online): 2446-1628
ISBN (online): 978-87-7573-740-6

AALBORG UNIVERSITY PRESS