Randomized reachability analysis in UPPAAL: fast error detection in timed systems*

Kiviriga, Andrej; Larsen, Kim Guldstrand; Nyman, Ulrik

Published in:
International Journal on Software Tools for Technology Transfer

DOI (link to publication from Publisher):
10.1007/s10009-022-00681-z

Publication date:
2022

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Kiviriga, A., Larsen, K. G., & Nyman, U. (2022). Randomized reachability analysis in UPPAAL: fast error detection in timed systems*. *International Journal on Software Tools for Technology Transfer*, *24*(6), 1025-1042. https://doi.org/10.1007/s10009-022-00681-z

Randomized Reachability Analysis in Uppaal: Fast Error Detection in Timed Systems*

Andrej Kiviriga, Kim Guldstrand Larsen and Ulrik Nyman

Department of Computer Science, Aalborg University, Street, Aalborg, 9220, Denmark.

*Corresponding author(s). E-mail(s): kiviriga@cs.aau.dk;
Contributing authors: kgl@cs.aau.dk; ulrik@cs.aau.dk;

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.

Keywords: model-checking, randomized, state space explosion, schedulability analysis, timed automata, stopwatch automata

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 property. 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 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 inheritance 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 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 [WCET, 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.

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

Table 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

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 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 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 randomized reachability analysis.
- 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 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 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 1 (Stopwatch Automaton[13]). *A Stopwatch Automaton (SWA) is represented as a tuple $A = (L, l_0, C, V, E, Act, I, D)$ 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 clocks can evolve at 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 2 (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

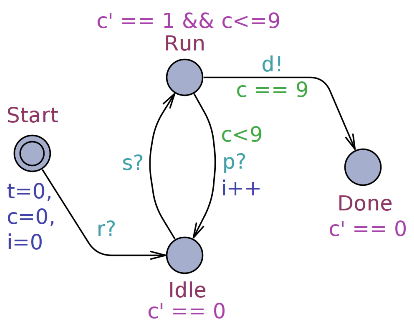


Fig. 1 Stopwatch Automaton example.

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 3. *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* $u' \models I(l')$.

An example of an arbitrary `Task` SWA is shown in Figure 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

The purpose of the randomized methods is to explore the state space quickly and be less affected by the state space explosion. The general pseudocode for the randomized reachability analysis is given in Algorithm 1. 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.

Table 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

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 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.

Algorithm 1 Randomized Reachability

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

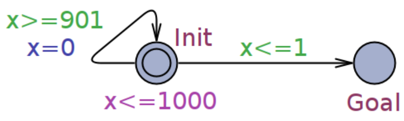


Fig. 2 Timed Automaton model with a `Goal` target state.

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 performed, i.e. how delays and transitions are chosen. The summary of the heuristics and their status is given in Table 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

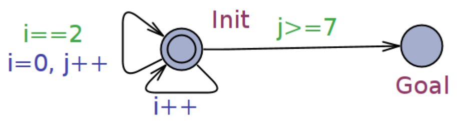


Fig. 3 Timed Automaton model of a difficult case for the RET heuristic.

require selecting a *target transition* first (line 14). The exact delay is then chosen only from that target transition's range of available delays (line 15). 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 14 and 15 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 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 14) 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 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.

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 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

Table 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%

traces have appeared to be spurious due to the over-approximate analysis of 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 15.

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 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.

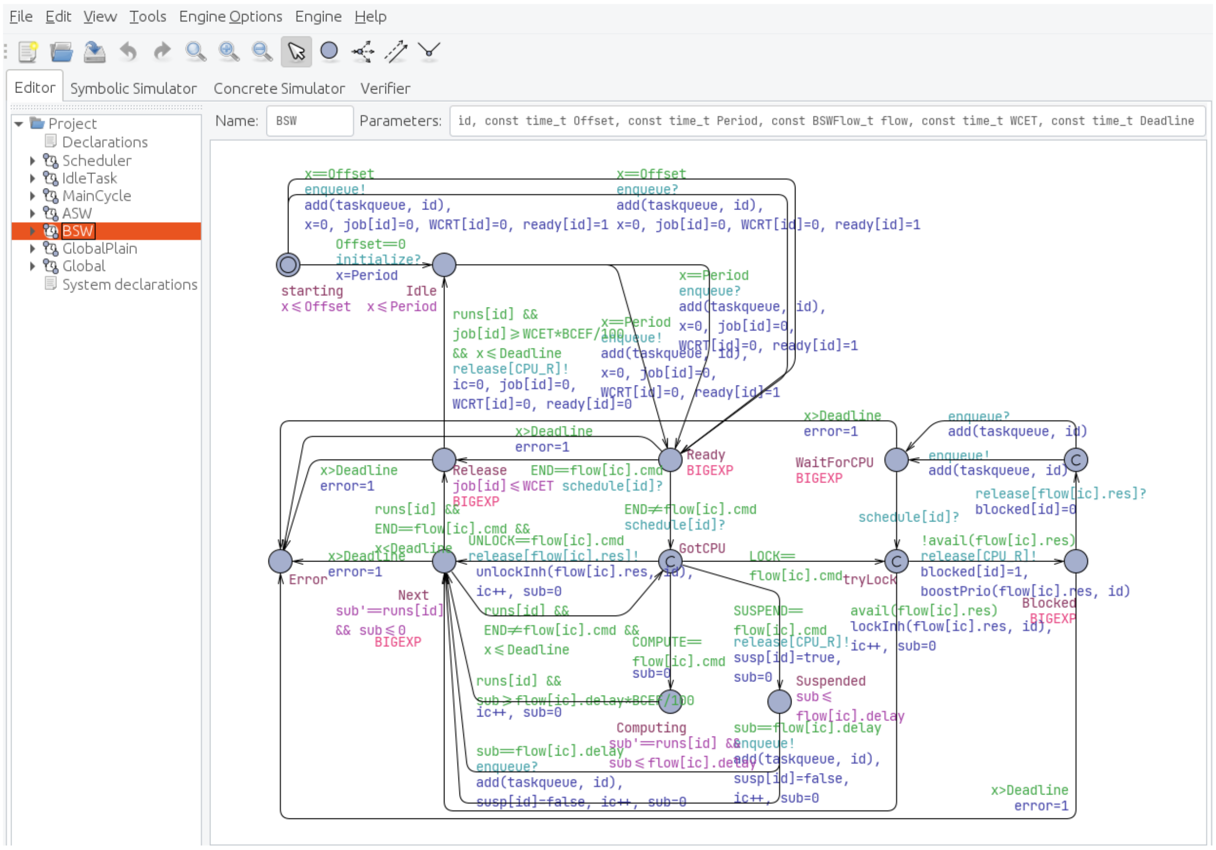


Fig. 4 The concrete simulator in UPPAAL

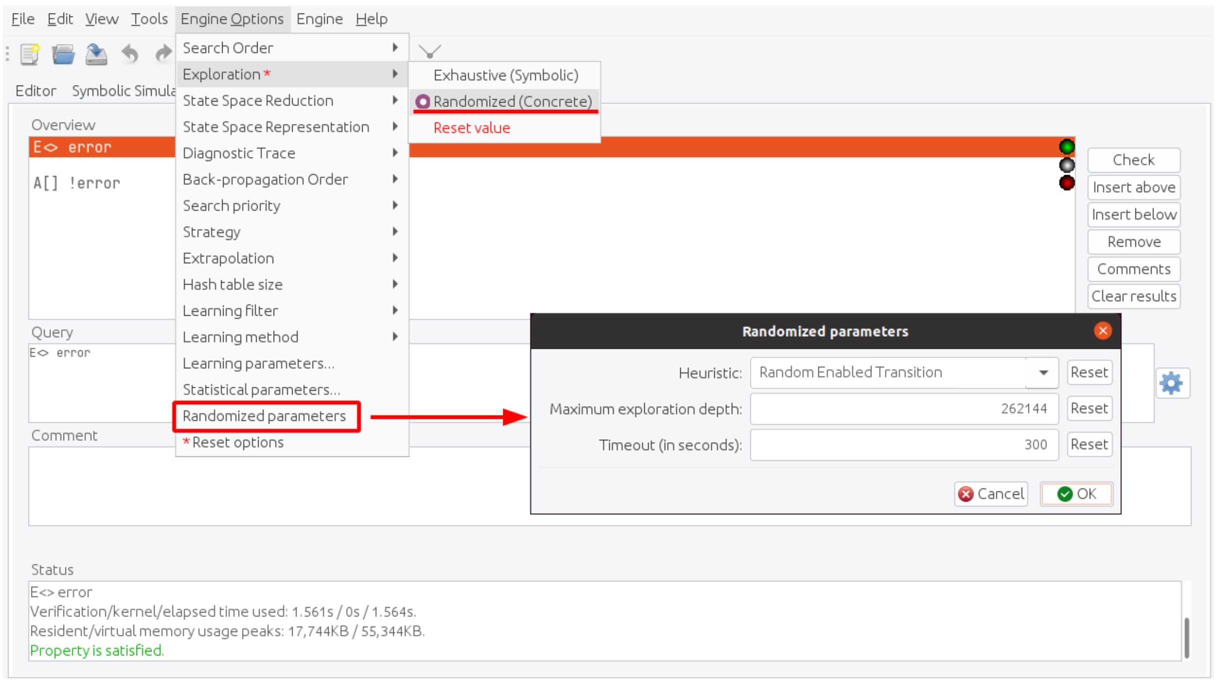
Shorter or Faster trace

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 16) and FireTransition (line 17) 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.

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



```
E<> error
A[] !error
```

Query: E<> error

Heuristic: Random Enabled Transition
Maximum exploration depth: 262144
Timeout (in seconds): 300

```
E<> error
Verification/kernel/elapsed time used: 1.561s / 0s / 1.564s.
Resident/virtual memory usage peaks: 17,744KB / 55,344KB.
Property is satisfied.
```

Fig. 5 The concrete simulator in UPPAAL

covered here and we refer the reader to [3] for a comprehensive presentation.

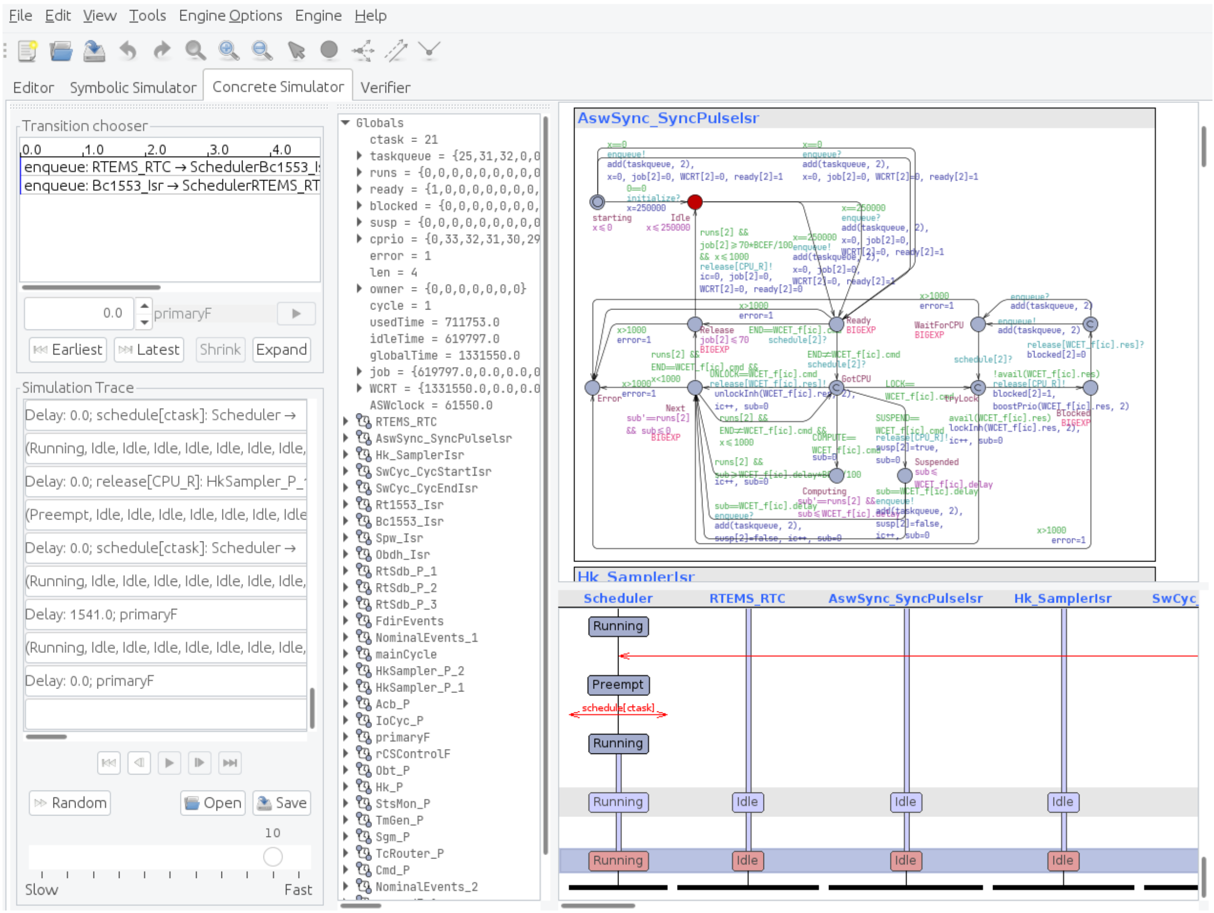
Figure 4 shows the editor tab of UPPAAL. This is the view used to create and edit the model of the system that is going to be analysed. On the left under the `Project` folder one can see all the templates that are part of the current model. In the right pane one of the templates, in this case the task 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 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 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 6 show the concrete simulator of UPPAAL, where concrete traces obtained from running Algorithm 1 (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



```
enqueue: RTEMS_RTC → SchedulerBc1553_I
enqueue: Bc1553_Isr → SchedulerRTEMS_RT
0.0  primaryF
```

```
Delay: 0.0; schedule[ctask]: Scheduler →
(Running, Idle, Idle, Idle, Idle, Idle, Idle, Idle,
Delay: 0.0; release[CPU_R]: HkSampler_P_
(Preempt, Idle, Idle, Idle, Idle, Idle, Idle, Idle
Delay: 0.0; schedule[ctask]: Scheduler →
(Running, Idle, Idle, Idle, Idle, Idle, Idle, Idle,
Delay: 1541.0; primaryF
(Running, Idle, Idle, Idle, Idle, Idle, Idle, Idle,
Delay: 0.0; primaryF
```

```
Globals
   ctask = 21
   taskqueue = {25,31,32,0,0
   runs = {0,0,0,0,0,0,0,0,0
   ready = {1,0,0,0,0,0,0,0,
   blocked = {0,0,0,0,0,0,0,
   susp = {0,0,0,0,0,0,0,0,0
   cprio = {0,33,32,31,30,29
   error = 1
   len = 4
   owner = {0,0,0,0,0,0,0,0}
   cycle = 1
   usedTime = 711753.0
   idleTime = 619797.0
   globalTime = 1331550.0
   job = {619797.0,0.0,0.0,0.0
   WCRT = {1331550.0,0.0,0.0,0
   ASWclock = 61550.0
RTEMS_RTC
AswSync_SyncPulseIsr
Hk_SamplerIsr
SwCyc_CycStartIsr
SwCyc_CycEndIsr
Rt1553_Isr
Bc1553_Isr
Spw_Isr
Obdh_Isr
RtSdb_P_1
RtSdb_P_2
RtSdb_P_3
FdirEvents
NominalEvents_1
mainCycle
HkSampler_P_2
HkSampler_P_1
Acb_P
IoCyc_P
primaryF
rCSControlF
Obt_P
Hk_P
StsMon_P
TmGen_P
Sgm_P
TcRouter_P
Cmd_P
NominalEvents_2
```

Fig. 6 The concrete simulator in UPPAAL

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 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 release of UPPAAL STRATEGO and can be found 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 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 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 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

Table 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

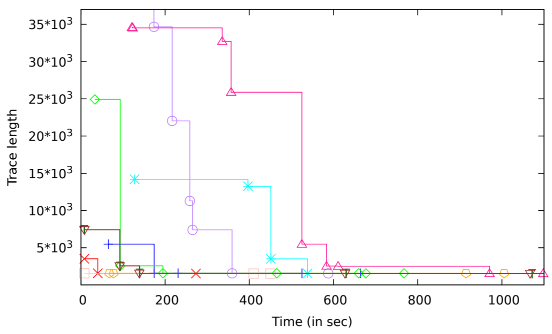


Fig. 7 10 runs of RET-S for Herschel with $f = 75\%$.

Table 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

hints at the possibility of the 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.

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 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 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 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 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.

We consider more schedulability systems modelled as stopwatch automata. Table 6 shows experiments for two different sets of schedulability problems: 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.

Table 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 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

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

Table 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

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 7 where each cell represents the average time of 100 runs, with the timeout of 2 hours 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

Table 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

with searching for a particular configuration of secrets in models with 6 girls and show results in Table 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 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 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 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 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 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 obvious 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

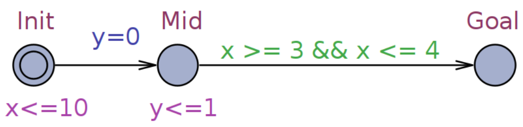


Fig. 8 Timed Automaton model of a difficult case for any heuristic.

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 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 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.

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 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.

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 nearly infeasible for 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] Kiviriga, A., Larsen, K.G., Nyman, U.: Randomized Refinement Checking of Timed I/O Automata. In: Pang, J., Zhang, L. (eds.) Dependable Software Engineering. Theories, Tools, and Applications, pp. 70–88. Springer, Cham (2020)

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

[3] Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: Formal Methods for the Design of Real-time Systems, pp. 200–236 (2004). Springer

[4] Joseph, M., Pandya, P.: Finding Response Times in a Real-Time System. The Computer Journal **29**(5), 390–395 (1986) https://academic.oup.com/comjnl/article-pdf/29/5/390/1314410/290390.pdf. https://doi.org/10.1093/comjnl/29.5.390

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

[6] Boudjadar, A., David, A., Kim, J., Larsen, K., Mikučionis, M., Nyman, U., Skou, A.: Statistical and exact schedulability analysis of hierarchical scheduling systems. Science of Computer Programming **127**, 103–130 (2016). https://doi.org/10.1016/j.scico.2016.05.008

[7] Boudjadar, A., David, A., Kim, J., Larsen, K., Mikučionis, M., Nyman, U., Skou, A.: A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling. Science of Computer Programming **113**(3), 236–260 (2015). https://doi.org/10.1016/j.scico.2015.10.003

[8] Brekling, A., Hansen, M.R., Madsen, J.: Moves — a framework for modelling and verifying embedded systems. In: 2009 International Conference on Microelectronics - ICM, pp. 149–152 (2009). https://doi.org/10.1109/ICM.2009.5418667

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

[10] David, A., Illum, J., Larsen, K.G., Skou, A.: Model-based framework for schedulability analysis using uppaal 4.1. Model-based design for embedded systems **1**(1), 93–119 (2009)

[11] David, A., Larsen, K.G., Legay, A., Mikučionis, M.: Schedulability of Herschel-Planck Revisited Using Statistical Model Checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies, pp. 293–307. Springer, Berlin, Heidelberg (2012)

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

[13] Cassez, F., Larsen, K.: The impressive power of stopwatches. In: Palamidessi, C. (ed.) CONCUR 2000 — Concurrency Theory, pp. 138–152. Springer, Berlin, Heidelberg (2000)

[14] Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation **205**(8), 1149–1172 (2007). https://doi.org/10.1016/j.ic.2007.01.009

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

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

[17] David, A., Larsen, K.G., Legay, A., Mikucionis, M., Poulsen, D.B.: Uppaal SMC tutorial. International Journal on Software Tools for Technology Transfer **17**(4), 397–415 (2015)

[18] Alur, R., Dill, D.: The theory of timed automata. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) Real-Time: Theory in Practice, pp. 45–73. Springer, Berlin, Heidelberg (1992)

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

[20] Larsen, K., Peled, D., Sedwards, S.: Memory-Efficient Tactics for Randomized LTL Model Checking. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments, pp. 152–169. Springer, Cham (2017)

[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 (2021). https://doi.org/10.1007/s10009-020-00597-6

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

[23] Martins Gomes, R., Baunach, M., Batista Ribeiro, L.: MCSmartOS: A Dependable OS for Compositional Embedded Systems. (2017). FoE-Tag des Field of Expertise "Information, Communication and Computing" ; Conference date: 28-03-2017

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

[25] Barbot, B., Basset, N., Beunardeau, M., Kwiatkowska, M.: Uniform sampling for timed automata with application to language inclusion measurement. In: Agha, G., Van Houdt, B. (eds.) Quantitative Evaluation of Systems, pp. 175–190. Springer, Cham (2016)

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

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