Randomized Refinement Checking of Timed I/O Automata*

Andrej Kiviriga, Kim Guldstrand Larsen, and Ulrik Nyman

Aalborg University, Selma Lagerløfs Vej 300, 9220 Aalborg, Denmark
{kiviriga, kgl, ulrik}@cs.aau.dk

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 memoryless 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.

Keywords: Model-checking · timed I/O automata · randomized · state-space · refinement.

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 [1,4,28]. Various symbolic and reduction techniques (e.g. [3,9,23,25,34]) 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. [22]) 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) [12] technique which has been intensely studied

*Supported by the ERC Advanced Grant Project: LASSO: Learning, Analysis, Synthesis and Optimization of Cyber-Physical Systems.

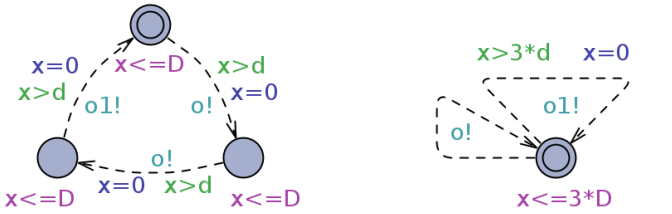
and applied to a variety of systems in model-checking including probabilistic systems [21], hybrid automata [37,31], Petri net state equations [35] and timed automata [27,20,29]. The core idea is to automatically generate *abstraction models* (e.g. by reducing the amount of clocks), which may have a 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 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 [11] 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 [19,30] 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 [24] 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 [18], where a *Deep Random Search* (DRS) algorithm, which explores the state-space of a simulation graph of a *timed automaton* (TA) [2] 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) [17].



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

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

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 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 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 [33,36]. 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 [13] which uses stochastic timed automata (STA). For more details of this stochastic semantics see [8,13]. Due to its simplicity, SMC is widely accepted in industrial applications where exhaustive model-checking is not feasible.

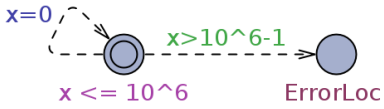


Fig. 2: Timed I/O Automaton. Difficult case for SMC.

For the purpose of violation discovery however, SMC simulation techniques may not be a productive approach. Figure 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 [16]. 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 1 (Timed I/O Automaton). *A Timed I/O Automaton (TIOA) is represented as 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 [16] for complete definition).

An example of **Researcher** TIOA, shown in Figure 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 cup of tea (`tea?`) the work can be done faster - between **6** and **7** time units. However, for

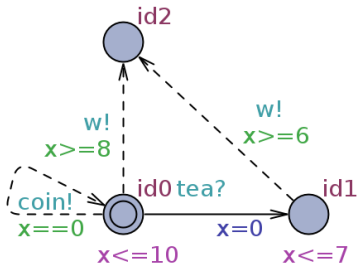


Fig. 3: Researcher automaton.

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`**:

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

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

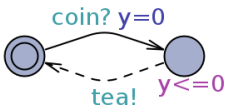


Fig. 4: **`Machine`** specification.

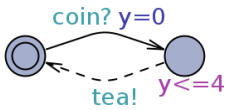
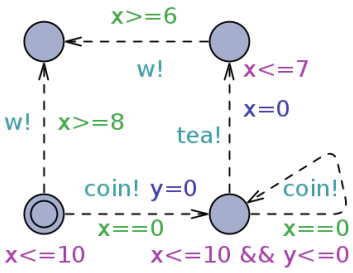
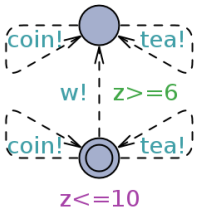


Fig. 5: **`SlowMachine`** specification.

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 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 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 [16] for formal definition). For illustration purposes, the automaton which captures the overall behavior of parallelly composed **`Machine`** and **`Researcher`** components is given in Figure 6 (a).



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



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

Fig. 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 2 (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 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 3 (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 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 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 [5] 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 [10]. 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 [10] 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 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 [7]). 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 to exclude 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 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 6 (b) ensures both upper and lower time limits to be followed by a more concrete system. Similarly in QuickCheck [11] 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.

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.

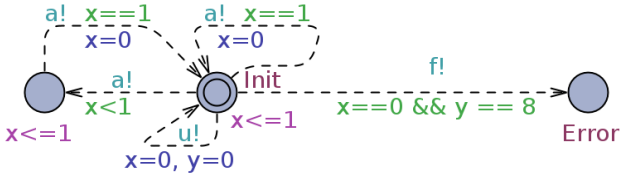


Fig. 7: Difficult case for RCF.

Consider the automaton from Figure 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 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 violation 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.

Table 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%

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;
```

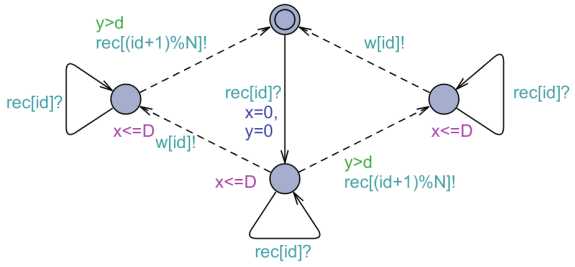
To fight this, we propose a strategy where each random walk has a different delay choice distribution, as shown in Algorithm 1. First, a random walk is executed where all the delays follow 50% LB / 50% UB distribution. If a violation is not found, a new random walk is issued where the probability to 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.

4 Test setting

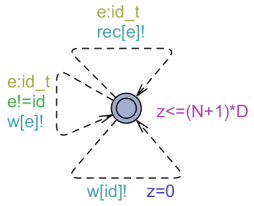
The experiments are performed on the models of Milner's scheduler [26] and Leader Election protocol [14], which operate on a ring topology and can be instantiated for an arbitrary number of nodes that communicate in sequence.

4.1 Milner's scheduler

We analyze a real-time version of Milner's scheduler [15], 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 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 [6]. 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.



(a) Node template.



(b) Overall specification template.

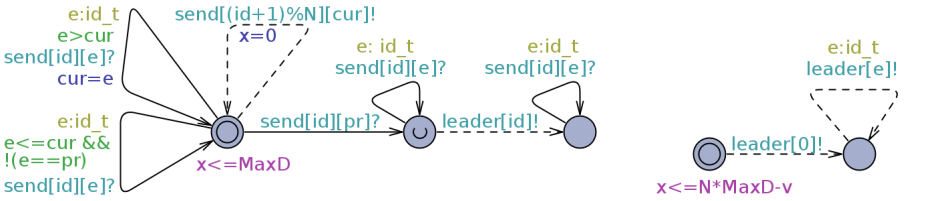
Fig. 8: Real-time version of Milner's Scheduler. Templates for the ECDAR tool.

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. $\{\mathtt{v} \in \mathbb{R} : 0 \leq \mathtt{v} \leq 1\}$ 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.

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.



(a) Node template.

(b) Overall specification template.

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

The templates for this protocol are shown in Figure 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. $\{\mathtt{v} \in \mathbb{Z} \colon 0 \leq \mathtt{v} \leq \mathtt{N*MaxD}\}$, 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 2 gives an overview on performance differences 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.

Table 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

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 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.

Table 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

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 counterexample 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 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

Table 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**

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.

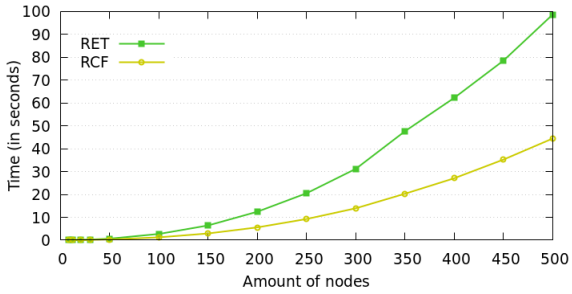


Fig. 10: RET and RCF comparison on Milner's scheduler with $d = 4$ and $v = 2\%$

In Figure 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 Evalua-

tion protocol. The results (shown in Table 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.

Table 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**

To further examine the efficiency of our heuristics to quickly detect violations during iterative development, we perform mutation testing on Leader Election protocol. Table 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 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).

Table 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	$M^{\exists}_1$	38.204	51.704	**11.697**
	$M^{\exists}_2$	25.183	0.002	**0.001**
	$M^{\exists}_3$	19.709	0.002	**0.001**
	$M^{\exists}_4$	18.007	0.002	**0.001**
N=6	$M^{\forall}_1$	12.592	0.077	**0.016**
	$M^{\forall}_2$	11.452	0.006	**0.001**
	$M^{\forall}_3$	10.643	0.003	**0.001**
	$M^{\forall}_4$	11.183	0.005	**0.001**

Settings		ECDAR	RET	RCF
N=7	$M^{\exists}_1$	nf	563.254	**125.991**
	$M^{\exists}_2$	nf	0.005	**0.001**
	$M^{\exists}_3$	nf	0.005	**0.001**
	$M^{\exists}_4$	687.573	0.005	**0.001**
N=7	$M^{\forall}_1$	nf	0.054	**0.011**
	$M^{\forall}_2$	nf	0.007	**0.001**
	$M^{\forall}_3$	nf	0.006	**0.001**
	$M^{\forall}_4$	35.230	0.001	**0.001**

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 [32]).

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. Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-Order Reduction in Symbolic State Space Exploration. In: Grumberg, O. (ed.) Computer Aided Verification. pp. 340–351. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
2. 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, Berlin, Heidelberg (1992)
3. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In: Halbwachs, N., Peled, D. (eds.) Computer Aided Verification. pp. 341–353. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
4. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and Upper Bounds in Zone Based Abstractions of Timed Automata. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 312–326. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
5. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for Playing Games! In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 121–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
6. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) 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. Lecture Notes in Computer Science, vol. 3185, pp. 200–236. Springer (2004). https://doi.org/10.1007/978-3-540-30080-9_7
7. Behrmann, G., Larsen, K.G., Pelánek, R.: To Store or Not to Store. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification. pp. 433–445. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
8. Bertrand, N., Bouyer, P., Brihaye, T., Carlier, P.: When are stochastic transition systems tameable? Journal of Logical and Algebraic Methods in Programming **99**, 41 – 96 (2018)
9. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.: Symbolic Model Checking: 1020 States and Beyond. Information and Computation **98**(2), 142 – 170 (1992)
10. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005 – Concurrency Theory. pp. 66–80. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
11. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. SIGPLAN Not. **46**(4), 53–64 (May 2011)
12. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
13. 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)
14. David, A., Larsen, K.G., Legay, A., Møller, M.H., Nyman, U., Ravn, A.P., Skou, A., Wasowski, A.: Compositional Verification of Real-Time Systems Using Ecdar. International Journal on Software Tools for Technology Transfer **14**, 703–720 (2012)

15. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In: Bouajjani, A., Chin, W.N. (eds.) Automated Technology for Verification and Analysis. pp. 365–370. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
16. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: 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. p. 91–100. HSCC '10, Association for Computing Machinery, New York, NY, USA (2010)
17. Dill, D.L.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems. pp. 197–212. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)
18. Grosu, R., Huang, X., Smolka, S.A., Tan, W., Tripakis, S.: Deep Random Search for Efficient Model Checking of Timed Automata. In: Kordon, F., Sokolsky, O. (eds.) Composition of Embedded Systems. Scientific and Industrial Issues. pp. 111–124. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
19. 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, Berlin, Heidelberg (2005)
20. He, F., Zhu, H., Hung, W.N., Song, X., Gu, M.: Compositional Abstraction Refinement for Timed Systems. In: 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering. pp. 168–176. IEEE (2010)
21. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. pp. 162–175. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
22. Kupferschmid, S., Wehrle, M., Nebel, B., Podelski, A.: Faster Than Uppaal? In: Gupta, A., Malik, S. (eds.) Computer Aided Verification. pp. 552–555. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
23. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction. In: Proceedings Real-Time Systems Symposium. pp. 14–24 (1997)
24. 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 International Publishing, Cham (2017)
25. Lind-Nielsen, J., Andersen, H.R., Behrmann, G., Hulgaard, H., Kristoifersen, K., Larsen, K.G.: Verification of Large State/Event Systems Using Compositionality and Dependency Analysis. In: Steffen, B. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 201–216. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
26. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag, Berlin, Heidelberg (1982)
27. Nagaoka, T., Okano, K., Kusumoto, S.: An Abstraction Refinement Technique for Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop. IEICE Trans. Inf. Syst. **93-D**(5), 994–1005 (2010). https://doi.org/10.1587/transinf.E93.D.994
28. Norris IP, C., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design **9**(1), 41–75 (1996)
29. Okano, K., Bordbar, B., Nagaoka, T.: Clock Number Reduction Abstraction on CEGAR Loop Approach to Timed Automaton. In: 2011 Second International Conference on Networking and Computing. pp. 235–241. IEEE (2011)

30. Oudinet, J., Denise, A., Gaudel, M.C., Lassaigne, R., Peyronnet, S.: Uniform Monte-Carlo Model Checking. In: Giannakopoulou, D., Orejas, F. (eds.) Fundamental Approaches to Software Engineering. pp. 127–140. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
31. Prabhakar, P., Duggirala, P.S., Mitra, S., Viswanathan, M.: Hybrid automata-based CEGAR for rectangular hybrid systems. Formal Methods in System Design **46**(2), 105–134 (2015)
32. Rasmussen, J.I., Behrmann, G., Larsen, K.G.: Complexity in Simplicity: Flexible Agent-Based State Space Exploration. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 231–245. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
33. 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, Berlin, Heidelberg (2004)
34. Valmari, A.: A Stubborn Attack on State Explosion. In: Clarke, E.M., Kurshan, R.P. (eds.) Computer-Aided Verification. pp. 156–165. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
35. Wimmel, H., Wolf, K.: Applying CEGAR to the Petri Net State Equation. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 224–238. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
36. Younes, H.L.S., Simmons, R.G.: Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling. In: Brinksma, E., Larsen, K.G. (eds.) Computer Aided Verification. pp. 223–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
37. Zutshi, A., Deshmukh, J.V., Sankaranarayanan, S., Kapinski, J.: Multiple Shooting, CEGAR-Based Falsification for Hybrid Systems. In: Proceedings of the 14th International Conference on Embedded Software. EMSOFT '14, Association for Computing Machinery, New York, NY, USA (2014)