A Symbolic Algorithm for the Analysis of Robust Timed Automata

Piotr Kordy[2], Rom Langerak[1], Sjouke Mauw[2], and Jan Willem Polderman[1]

[1] University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands
[2] Université du Luxembourg, rue Richard Coudenhove-Kalergi 6, L-1359 Luxembourg, Luxembourg

Abstract. We propose an algorithm for the analysis of robustness of timed automata, that is, the correctness of a model in the presence of small drifts of the clocks. The algorithm is an extension of the region-based algorithm of Puri and uses the idea of stable zones as introduced by Daws and Kordy. Similarly to the assumptions made by Puri, we restrict our analysis to the class of timed automata with closed guards, progress cycles, and bounded clocks. We have implemented the algorithm and applied it to several benchmark specifications. The algorithm is a depth-first search based on on-the-fly reachability using zones.

1 Introduction

One of the most successful current paradigms for the specification and analysis of real-time systems is the timed automata model [2]. Timed automata are automata extended by clock variables that can be tested and reset. Numerous real-time systems have been specified and analysed using the Uppaal tool [3, 17] and the approach can be said to be mature and industrially applicable.

An important issue for timed automata specifications is robustness: what happens if there are small imprecisions in the clocks or in the tests on clocks? It appears that in that case more states are reachable, which means that a specification that has been proven to be correct may no longer be correct in the presence of imprecisions, even if they are arbitrarily small. Of course, this has disturbing implications for the implementation of systems, as in real systems imprecisions cannot be avoided. This important problem has been first addressed in the seminal work by Puri [18], later improved and extended by De Wulf et al. [11]. Their main result is the introduction of an *enlarged semantics* of timed automata. In the enlarged semantics all clocks are allowed to drift by a small (positive) perturbation ϵ, in order to model the imprecisions of the clocks. A timed automaton is said to be implementable if there exists a value for ϵ for which the behaviour of the timed automaton conforms to the specification under the enlarged semantics.

Robust model-checking has been solved for safety properties [18, 11] and for richer linear-time properties [4, 5]. The analysis and algorithms provided by these

papers are based on region automata. As the number of regions grows exponentially with the size of the largest constant used in a timed automaton, region-based algorithms are not really suitable for a practical implementation. What would be needed is a symbolic analysis in terms of zones, which form the fundamental concept for the implementation of a tool like Uppaal [3]. A first step in this direction has been provided by Daws and Kordy [8], who defined the notion of *stable zones* and related it to the region-based approach of Puri et al. [18, 10]. The algorithm of Daws and Kordy, which uses the concept of a stable zone, works only for flat timed automata, i.e., automata that have no nested cycles. This restriction significantly limits the practical usability of the algorithm. One solution could be to transform the timed automaton into a flat timed automaton using results from Comon and Jurski [7]. Unfortunately, the resulting flat timed automaton may be exponentially larger than the starting timed automaton. In this paper we show how the stable zone concept leads to a practical implementation. We propose a fully symbolic algorithm to solve the robust reachability problem. To validate practical usability of the algorithm, we implemented a simple tool and performed a number of experiments.

Another solution to the robustness problem was suggested by Dima [13], who proposed to combine the symbolic reachability algorithm with cycle detection and expanding borders of the zones. That algorithm is an improvement over the purely region-based algorithm of Puri. But at one point it looks for bordering regions to the reachable set of states and checks if they are on a strongly connected component. So, similarly to the region-based algorithm, the running time may depend on the size of the constants used in the timed automaton.

The rest of the paper is structured as follows: in Section 2 we provide the necessary background on timed automata, extended semantics, and stable zones. In Section 3 we present a reachability algorithm based on stable zones, which is proven correct in Section 4. Section 5 contains the results of experiments with the implementation, and Section 6 contains the conclusions and perspectives.

2 Preliminaries

2.1 Timed Automata (TA)

Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be a set of variables, called clocks. In this work we will only consider bounded clocks, meaning that there is an upper bound $M \in \mathbb{N}$ on the clock values. A *clock valuation* is a function $v : \mathcal{X} \mapsto [0, M] \subset \mathbb{R}$, which assigns to each clock a non-negative value $v(x)$ that is smaller than or equal to this upper bound. By $\mathbb{R}^{\mathcal{X}}_{\geq 0}$ we denote the set of all valuations over $\mathcal{X}$.

Definition 1 (Closed Zones). *A* closed zone *over a set of clocks $\mathcal{X}$ is a set of clock valuations that satisfy constraints defined by the grammar $g ::= x \rhd d \mid x - y \rhd d \mid g \wedge g$, where $x, y \in \mathcal{X}$, $d \in \mathbb{Z}$ and $\rhd \in \{\leq, \geq\}$. The set of closed zones over $\mathcal{X}$ is denoted by $\mathcal{Z}(\mathcal{X})$.*

To simplify notation, we will often simply write the constraint itself to denote the set of clock valuations that it implies. Further, we will use familiar notation,

like $a \leq x \leq b$, to denote composite constraints, such as $(a \leq x) \wedge (x \leq b)$. We will often write $\mathcal{Z}$ instead of $\mathcal{Z}(\mathcal{X})$ if $\mathcal{X}$ can be derived from the context. A *rectangular zone* Z is a closed zone with no bounds on clock differences, i.e., with no constraints of the form $x - y \rhd d$. The set of rectangular zones over $\mathcal{X}$ is denoted by $\mathcal{Z}_R(\mathcal{X})$. The set $\mathcal{Z}_U(\mathcal{X})$ denotes the set of *upper zones*, that is, rectangular zones with no lower bounds on the clocks.

Definition 2 (TA). *A* timed automaton *[2] is a tuple* $A = (\mathcal{X}, Q, I, q_0, \mathcal{E})$ *where*
- $\mathcal{X}$ *is a finite set of clocks,*
- Q *is a set of* locations*,*
- $I \colon Q \to \mathcal{Z}_U(\mathcal{X})$ *is a function that assigns to each location an invariant* $I(q)$*,*
- $q_0 \in Q$ *is the* initial location*,*
- $\mathcal{E}$ *is a finite set of* edges*. An edge is a tuple of the form* $e = (q, Z_g, X, q')$*, where* q, q' *are the source and target locations,* $Z_g \in \mathcal{Z}_R(\mathcal{X})$ *is an enabling guard and* $X \subseteq \mathcal{X}$ *is the set of clocks to be reset.*

We use the word *location* to denote a node of the automaton, rather than the more commonly used word *state*, which we will use in the semantics to denote a pair of a location and a valuation. An example of a timed automaton can be seen in Figure 1. The set of clocks is $\mathcal{X} = \{x_1, x_2, x_3\}$ and the set of locations is $Q = \{q_1, q_2, q_3\}$. The initial location is q_1. The arrows represent the set of edges $\mathcal{E}$. For example in the edge from q_1 to q_2 the guard requires $2 \leq x_1 \leq 4$, and, when taking this edge, clock x_1 will be reset ($X = \{x_1\}$). In this example, we ignored the potential use of invariants.

2.2 Semantics

The semantics of a timed automaton is defined as a transition system, where a *state* $(q, v) \in Q \times \mathbb{R}_{\geq 0}^{\mathcal{X}}$ consists of the current location and the current values of the clocks. There are two types of transitions between states: the automaton may either delay for some time (a delay transition), or follow an enabled edge (an action transition) while resetting some clocks.

For $t \in \mathbb{R}$, we define the valuation $v + t$ as follows: for all $x \in \mathcal{X}$, $(v + t)(x) = v(x) + t$. This expression is only defined if for all clocks x, $0 \leq v(x) + t \leq M$. We sometimes write $v - t$ for $v + (-t)$. For $X \subseteq \mathcal{X}, v[X := 0]$ is the valuation such that $v[X := 0](x) = 0$, for $x \in X$, and $v[X := 0](x) = v(x)$, for $x \in \mathcal{X} \setminus X$. We denote the valuation that assigns 0 to every clock by $\mathbf{0}$.

Definition 3 (Standard semantics). *The* standard semantics *of a timed automaton* $A = (\mathcal{X}, Q, I, q_0, \mathcal{E})$ *is a transition system* $[A] = (\mathcal{S}, \to)$*, where states are pairs* $(q, v) \in \mathcal{S}$*, with* $q \in Q$*,* $v \in I(q)$*. The initial state is* $s_0 = (q_0, \mathbf{0})$*, i.e., the initial location with all clocks equal to zero. For* $t \in \mathbb{R}_{\geq 0}$ *and* $e \in \mathcal{E}$*, the transition relation* $\cdot \xrightarrow{\cdot} \cdot \colon \mathcal{S} \times (\mathbb{R}_{\geq 0} \cup \mathcal{E}) \times \mathcal{S}$ *is defined by:*
- $(q, v) \xrightarrow{t} (q, v + t)$*, if* $v + t \in I(q)$ *,* $t \in \mathbb{R}_{\geq 0}$*,*
- $(q, v) \xrightarrow{e} (q', v[X := 0])$*, if* $e = (q, Z_g, X, q') \in \mathcal{E}$*,* $v \in Z_g$*,* $v[X := 0] \in I(q')$*.*

Given a timed automaton, the interesting question is which states are reachable from the initial state. Reachable states may be used to characterize safety properties of a system. Formally, the set of reachable states of a timed automaton is the smallest set $\mathcal{U} \subseteq \mathcal{S}$ of states containing $s_0 = (q_0, \mathbf{0})$, and satisfying the following condition: if $(q, v) \xrightarrow{t} (q', v')$ for some $t \in \mathbb{R}_{\geq 0}$ or $(q, v) \xrightarrow{e} (q', v')$ for some $e \in \mathcal{E}$, and $(q, v) \in \mathcal{U}$, then $(q', v') \in \mathcal{U}$.

Let us look again at the example in Figure 1. To simplify notation, we will denote the values for the consecutive clocks by a vector $[x_1, x_2, x_3]$. The initial state is $(q_1, [0, 0, 0])$. Here is an example of a sequence of states implied by the standard semantics $[A]$:

$$(q_1, [0, 0, 0]) \xrightarrow{2} (q_1, [2, 2, 2]) \xrightarrow{e_1} (q_2, [0, 2, 2]) \xrightarrow{e_2} (q_3, [0, 0, 2]) \xrightarrow{e_3} (q_1, [0, 0, 0]),$$

where e_i denotes an edge from a location q_i. Note that this is the only possible sequence that will get us back into the initial location (disregarding splitting the first transition into separate delay steps). As we can see, it is not possible to reach the *Err* location using the standard semantics $[A]$.

2.3 Symbolic Semantics

In general, the semantics $[A]$ has a non-countable number of states. To be able to reason about these states, abstractions are used which are based on zones [14]. This results in a symbolic semantics where a symbolic state S is a pair $(q, Z) \in Q \times \mathcal{Z}$ consisting of a location and a zone.

We define time passing for zones: $\uparrow Z = \{v + t \in \mathbb{R}^{\mathcal{X}}_{\geq 0} \mid v \in Z \wedge t \in \mathbb{R}_{\geq 0}\}$. Similarly, we define $\downarrow Z = \{v - t \in \mathbb{R}^{\mathcal{X}}_{\geq 0} \mid v \in Z \wedge t \in \mathbb{R}_{\geq 0}\}$. It can be shown that the set of zones $\mathcal{Z}(\mathcal{X})$ is closed under these operations.

Definition 4 (Symbolic semantics). *The* symbolic semantics $[\![A]\!]$ *is a transition system* $(Q \times \mathcal{Z}, \Rightarrow)$ *with initial state* $(q_0, \{\mathbf{0}\})$*, where* $\{\mathbf{0}\}$ *is the zone in which all clocks are equal to zero. For* $e \in \mathcal{E}$*, transitions are defined by the rules:*

- $(q, Z) \overset{\uparrow}{\Rightarrow} (q, I(q) \cap \uparrow Z)$,
- $(q, Z) \overset{e}{\Rightarrow} (q', Z')$*, where* $e = (q, Z_g, X, q')$ *is an edge and* $Z' = \{v' \mid \exists v \in Z, (q, v) \xrightarrow{e} (q', v')\} \neq \emptyset$.

To differentiate between states of standard and symbolic semantics, we will refer to the latter as *symbolic states*. Such a symbolic state can be interpreted as a set of states. Given state $s = (q, x)$ and symbolic state $S = (q', Z)$ when $q = q'$ and $x \in Z$ we will abuse notation and write $s \in S$. Similarly, given $S = (q, Z)$ and $S' = (q, Z')$ we say that $S \subseteq S'$ when $Z \subseteq Z'$ and $\uparrow S = (q, \uparrow Z)$.

For timed automata with bounded clocks, the symbolic semantics defined above is sound, complete and finite [2]. For timed automata with unbounded clocks, extrapolation can be used to ensure finiteness of the symbolic semantics [3]. To illustrate the symbolic semantics, we show a sequence of symbolic

states of the timed automaton from Figure 1.

$$(q_1, \{x_1 = x_2 = x_3 = 0\}) \overset{\uparrow}{\Rightarrow} (q_1, \{x_1 = x_2 = x_3\}) \overset{e_1}{\Rightarrow}$$
$$(q_2, \{x_1 = 0, x_2 = x_3, 2 \leq x_2 \leq 4\}) \overset{\uparrow}{\Rightarrow} (q_2, \{x_2 = x_3, 2 \leq x_2 - x_1 \leq 4\}) \overset{e_2}{\Rightarrow}$$
$$(q_3, \{x_1 = x_2 = 0, x_3 = 2\}) \overset{\uparrow}{\Rightarrow} (q_3, \{x_1 = x_2 = x_3 - 2\}) \overset{e_3}{\Rightarrow} (q_1, \{x_1 = x_2 = x_3 = 0\})$$

Note that the *Err* location is again not reachable.

2.4 Extended Semantics

The semantics of timed automata makes unrealistic assumptions because it requires instant reaction time and clocks that are infinitely precise. To remedy these limitations, we give the parametric semantics introduced by Puri [18] that enlarges the (normal) semantics of timed automata. This semantics can be defined in terms of timed automata, extended with a small, positive, real valued parameter, denoted ϵ.

Definition 5 (Extended semantics.). *Given parameter ϵ, the* extended semantics $[A]^\epsilon$ *is a transition system $(\mathcal{S}, \rightsquigarrow_\epsilon)$ with initial state $s_0 = (q_0, \mathbf{0})$. For $t \in \mathbb{R}_{\geq 0}$ and $e \in \mathcal{E}$, transitions are defined by the rules:*

- $(q, v) \overset{t}{\rightsquigarrow}_\epsilon (q, v')$ *if* $v' \in I(q)$ *and* $\forall x \in \mathcal{X}\ v'(x) - v(x) \in [(1-\epsilon)t, (1+\epsilon)t]$,
- $(q, v) \overset{e}{\rightsquigarrow}_\epsilon (q', v[X := 0])$, *if* $e = (q, Z_g, X, q') \in \mathcal{E}$, $v \in Z_g$, $v[X := 0] \in I(q')$.

Note that the above definition allows perturbation to grow with time. If we substitute ϵ by 0, then we obtain the standard semantics. Let $\mathcal{S}^\epsilon$ denote the set of states reachable using transitions $\rightsquigarrow_\epsilon$ in extended semantics $[A]^\epsilon$. Clearly, the set of reachable states $\mathcal{U} \subseteq \mathcal{S}$ in standard semantics $[A]$ is a subset of $\mathcal{S}^\epsilon$. Calculating $\mathcal{S}^\epsilon$ is undecidable for more than two clocks [20]. As a solution, Puri has proposed to calculate the set $\mathcal{S}^* = \lim_{\epsilon \to 0} \mathcal{S}^\epsilon$, which equals $\bigcap_{\epsilon > 0} \mathcal{S}^\epsilon$, and represents the set of states that are reachable for an arbitrarily small ϵ.

Looking again at the example in Figure 1, for any small $\epsilon > 0$ we have the following sequence of states:

$$(q_1, [0,0,0]) \overset{2}{\rightsquigarrow}_\epsilon (q_1, [2, 2-2\epsilon, 2-2\epsilon]) \overset{e_1}{\rightsquigarrow}_\epsilon (q_2, [0, 2-2\epsilon, 2-2\epsilon]) \overset{2\epsilon}{\rightsquigarrow}_\epsilon (q_2, [2\epsilon, 2, 2])$$
$$\overset{e_2}{\rightsquigarrow}_\epsilon (q_3, [2\epsilon, 0, 2]) \overset{e_3}{\rightsquigarrow}_\epsilon (q_1, [2\epsilon, 0, 0]) \overset{2-2\epsilon}{\rightsquigarrow}_\epsilon (q_1, [2, 2-4\epsilon+2\epsilon^2, 2-4\epsilon+2\epsilon^2]) \rightsquigarrow_\epsilon$$
$$\ldots \rightsquigarrow_\epsilon (q_1, [4\epsilon, 0, 0]) \rightsquigarrow_\epsilon \ldots \rightsquigarrow_\epsilon (q_1, [n2\epsilon, 0, 0]), \text{ where } n \in \mathbb{N}_{\geq 2}.$$

Note that after one cycle we reach the state $(q_1, [2\epsilon, 0, 0])$, which is not reachable in $[A]$. Following the same sequence of edges n times allows us to accumulate the small imprecision and reach state $(q_1, [n2\epsilon, 0, 0])$. For any value ϵ we can find sufficiently large n such that $n2\epsilon \geq 2$, so the *Err* location is reachable in $\mathcal{S}^*$.

Puri's approach [18] to calculate $\mathcal{S}^*$ is based on the concept of a *region*. We will briefly explain his approach and indicate possibilities for improvement. For arbitrarily small ϵ, the geometrical distance between clock valuations in

$(q_1, [n2\epsilon, 0, 0])$ and $(q_1, [(n+1)2\epsilon, 0, 0])$ is small. Intuitively, if two clock valuations are *close enough*, we say that states are in the same *region*. By *close enough* we mean that they give to each clock the same integral part and when the clocks are sorted according to the fractional part of their valuation, they will form the same sequence of clocks. Consequently, regions form an equivalence relation on the set of states. Regions are interesting because states in the same region will give rise to similar behaviours, meaning that the same transitions are available. The *region graph* is a graph where the nodes are regions and there exists an edge between regions c_1 and c_2 if $\exists s_1 \in c_1\, s_2 \in c_2$ such that $s_1 \to s_2$. The states $(q_1, [n2\epsilon, 0, 0])$ and $(q_1, [(n+1)2\epsilon, 0, 0])$ would be on a cycle in the region graph. For a precise description of region graphs we refer to [2, 3].

Puri [18] shows that being on a cycle in a region graph is a necessary and sufficient condition to accumulate errors due to clock drift. Based on this observation, he proposes an algorithm to calculate $\mathcal{S}^*$. The algorithm does normal reachability analysis and adds regions that are on a cycle in a region graph and have some common part with the set of states calculated so far. To avoid the problem of an infinite number of possible cycles in the region graph, he uses strongly connected components.

This way, Puri reduced the problem of finding the set $\mathcal{S}^*$ to the problem of finding all reachable strongly connected components on the region graph of a timed automaton and calculating normal reachability, and thus he does not need to use extended semantics in his algorithm.

Though of conceptual value, this algorithm is not suitable for implementation, since the number of regions is exponentially large: $O\big(|\mathcal{X}|!M^{|\mathcal{X}|}\big)$ [2]. In order to obtain a practical, more efficient robustness algorithm, we need an analysis in terms of zones [21].

To summarize, we have introduced standard semantics, which has an infinite number of possible states. The symbolic semantics uses zones to reduce the number of states to a finite amount of abstract states and this is the semantics used in any practical implementation. The last one is the extended semantics that allows the clocks to drift in time, which is undecidable in general for a given value of ϵ, but which can be calculated if ϵ is infinitely small. In the rest of the paper we will show how to calculate $\mathcal{S}^*$ using abstractions based on zones rather than on regions.

2.5 Stable Zones

In this section we briefly recall the most important notions concerning the concept of *stable zones* as introduced in [8]. Note that a stable zone is a set of states, not a set of valuations. Stable zones are defined for *edge cycles*.

Definition 6 (Edge cycle). *An* edge cycle *of a timed automaton A is a finite sequence of edges $\sigma = e_1 \dots e_k$ such that the source location of e_1 is the same as the destination location of e_k. A* progress cycle *is an edge cycle where each clock is reset at least once.*

Following the assumptions in previous work on robustness [18, 11, 8], we will only consider automata in which all edge cycles are progress cycles.

To simplify notation, given a sequence of edges $\sigma = e_1 \dots e_k$, we will write $\overset{\sigma}{\Rightarrow}$ for $\overset{\uparrow}{\Rightarrow}\overset{e_1}{\Rightarrow}\overset{\uparrow}{\Rightarrow}\overset{e_2}{\Rightarrow}\overset{\uparrow}{\Rightarrow} \dots \overset{\uparrow}{\Rightarrow}\overset{e_k}{\Rightarrow}\overset{\uparrow}{\Rightarrow}$ and $\overset{\sigma}{\rightarrow}=\overset{t_0}{\rightarrow}\overset{e_1}{\rightarrow}\overset{t_1}{\rightarrow} \dots \overset{e_k}{\rightarrow}\overset{t_k}{\rightarrow}$ for any $t_0, t_1 \dots t_k \in \mathbb{R}_{\geq 0}$.

Definition 7 (Stable zone). *A* stable zone *for an edge cycle σ in a timed automaton is the largest set of states $W_\sigma \subseteq Q \times \mathbb{R}^{\mathcal{X}}_{\geq 0}$ such that*

$$\forall s \in W_\sigma \, \exists s_1, s_2 \in W_\sigma : s_1 \overset{\sigma}{\rightarrow} s \overset{\sigma}{\rightarrow} s_2.$$

Thus, a stable zone W_σ is a set of states such that if we cycle (forward or backward) along an edge cycle σ we have the possibility to stay inside W_σ. Intuitively this allows infinite cycling and as a consequence arbitrary accumulation of small imprecisions in clock drifts. This has been formally shown in [8], which can be summarized in the following lemma:

Lemma 1. *For any $s, s' \in W_\sigma$ and for any $\epsilon > 0$, $s \rightsquigarrow_\epsilon \rightsquigarrow_\epsilon \dots \rightsquigarrow_\epsilon s'$.*

Lemma 1 states that, given an arbitrarily small ϵ, starting from any state in W_σ, we can reach any other state in W_σ in the extended semantics $[A]^\epsilon$. Therefore, during extended reachability analysis, we can add stable zones as a whole in a similar way as Puri's algorithm adds strongly connected components of the region graph.

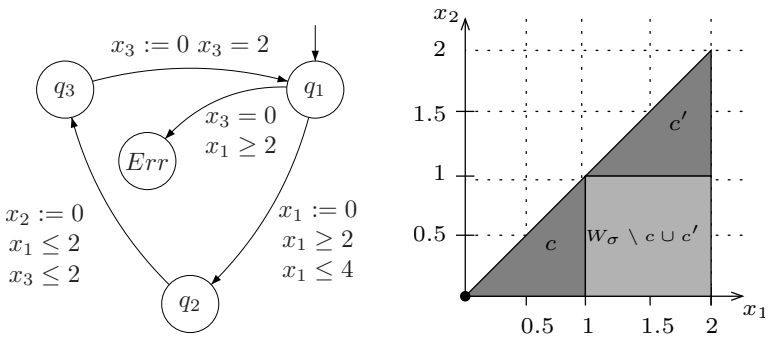

Fig. 1. Timed automaton (left) and graph of $\mathcal{S}^*$(right). The graph shows a part of $\mathcal{S}^*$ for location q_1 and clock value $x_3 = 0$.

Looking at the example in Figure 1, we can see a timed automaton on the left side. On the right side, the three grey areas represent the set $\mathcal{S}^*$ for location q_1 and clock value $x_3 = 0$. For this clock value, the only reachable state in standard semantics on the graph is $(q_1, [0, 0, 0])$. In order to illustrate Puri's algorithm, we divided the grey area in three subareas. The areas c and c' are regions that are on a cycle in the region graph, meaning that from any state in c, we can reach some other state c by taking at least one edge transition. For example $(q_1, [0.5, 0.2, 0]) \overset{e_1 e_2 e_3}{\rightarrow} (q_1[0.5, 0.5, 0])$.

Puri's algorithm would calculate $\mathcal{S}^*$ in the following way: state $(q_1, [0,0,0])$ would be added by standard reachability. Next, region c would be added because it is on a cycle in the region graph. After that, region c' would be added because $c \cap c' \neq \emptyset$ and c' is also on a cycle. The lighter shaded area would be added at the end, by checking what states are reachable from newly added regions. In contrast, the algorithm from [8] would add the whole grey area in one step because it lies inside a stable zone.

To show how to calculate stable zones in an efficient way, we introduce some additional notation. Let $e = (q, Z_g, X, q')$ be an edge. We define $\text{post}_e((q, Z)) = (q', \uparrow Z')$, where $(q, Z) \stackrel{e}{\Rightarrow} (q', Z')$. In other words, for $S \in Q \times \mathcal{Z}$, $\text{post}_e(S)$ is a symbolic state that contains all states that can be reached from the state S by taking edge e and later allowing time to pass. Similarly, we define $\text{pre}_e((q', Z)) = (q, \downarrow Z')$, where $Z' = \{v \mid \exists v' \in Z, (q, v) \stackrel{e}{\rightarrow} (q', v')\}$. For sequence of edges $\sigma = e_1 \dots e_n$, we define as $\text{post}_\sigma(S) = \text{post}_{e_n}(\dots \text{post}_{e_1}(S))$ and $\text{pre}_\sigma(S) = \text{pre}_{e_n}(\dots \text{pre}_{e_1}(S))$.

The following lemma from [8] gives a feasible algorithm to calculate stable zones as a fixpoint:

Lemma 2. $W_\sigma = \nu S.(post_\sigma(S) \cap pre_\sigma(S)) = (\nu S.post_\sigma(S)) \cap (\nu S.pre_\sigma(S))$.

The ν operator is the greatest fixpoint operator from μ-calculus. We need this operator because a stable zone is defined as a maximal set. Intuitively, we need post_σ and pre_σ to ensure existence of predecessors and successors for any state in W_σ. The algorithm proposed in [8] starts from the idea that all stable zones are calculated a priori, on the basis of all edge cycles σ in the timed automaton. Thus, the approach works only for flat timed automata (automata without nested cycles). To extend the algorithm to non-flat timed automata is not trivial. For example, it is not enough to consider only minimal edge cycles. It is possible that two edge cycles σ_1 and σ_2 both have an empty stable zone, whereas the edge cycle $\sigma_1\sigma_2$ has a non-empty stable zone. In the next section we present an algorithm that uses the concept of stable zones. but calculates them using fixpoint calculation.

3 A Symbolic Algorithm for the Extended Semantics

The purpose of the algorithm is to calculate the reachability relation for the extended semantics. It means that, given a state Goal and an initial state S_0, the algorithm will check if the Goal state is reachable for an arbitrarily small value of $\epsilon > 0$ under extended semantics $[A]^\epsilon$. This can be achieved by performing a normal reachability analysis in $[\![A]\!]$ while ensuring that all reachable stable zones are added. We will detect that we touched a potential stable zone when we reach a symbolic state that we have seen before. Given a symbolic state potentially touching a stable zone (or zones), we need to do the greatest fixed point calculation, described in detail in Section 3.2 (function `AllSZones`).

To make the algorithm more efficient we try to limit the set of states/edges for which we have to do the fixpoint calculation by grouping locations together

into *strongly connected sets* – $\mathsf{SCS}_{\mathcal{U}}$). $\mathsf{SCS}_{\mathcal{U}}$ is a minimal set of locations with the property that, if we start with any reachable state in these locations and follow any path that returns to the same location, we will not visit any location outside $\mathsf{SCS}_{\mathcal{U}}$. Given $e = (q, Z_g, X, q')$ we say that $e \vDash \mathsf{SCS}_{\mathcal{U}}$ if $q \in \mathsf{SCS}_{\mathcal{U}}$ and $q' \in \mathsf{SCS}_{\mathcal{U}}$. Similarly $\sigma \vDash \mathsf{SCS}_{\mathcal{U}}$ if $\sigma = e_1 \dots e_n$ and $\forall_{1 \leq i \leq n} e_i \vDash \mathsf{SCS}_{\mathcal{U}}$.

Definition 8 (Strongly Connected Set – $\mathsf{SCS}_{\mathcal{U}}$). *Let $\mathcal{U} \subseteq \mathcal{S}$ be a set of states in $[A]$. $\mathsf{SCS}_{\mathcal{U}} \subseteq Q$ is the minimal set of locations q, such that $\forall (q, v), (q, v') \in \mathcal{U}$ such that $(q, v) \xrightarrow{\sigma} (q, v')$ it holds that $\sigma \vDash \mathsf{SCS}_{\mathcal{U}}$, and $\forall (q, v) \in \mathcal{U}$: $q \in \mathsf{SCS}_{\mathcal{U}}$*

To find all such $\mathsf{SCS}_{\mathcal{U}}$ sets, we will combine the depth-first search reachability algorithm for timed automata and Tarjan's algorithm [19] to find strongly connected components.

3.1 The Main algorithm

We use the following notation: $\mathcal{U}, \mathcal{U}', \mathcal{U}_{curr}, \mathcal{U}_{prev}, \mathcal{U}_{seen} \subseteq Q \times \mathcal{Z}$, are sets of symbolic states; $S, S', S'', S_0, \mathsf{Goal} \in Q \times \mathcal{Z}$ are symbolic states; $Q_{\text{mark}}, \mathsf{SCS}_{\mathcal{U}} \subseteq Q$ is a set of locations; $q \in Q$ is a location; ST,Open are stack structures holding symbolic states. We can do typical operations on stack structures: $\mathsf{ST}.\mathtt{push}(S)$ will add S at the top of ST, and $S := \mathsf{ST}.\mathtt{pop}()$ will remove the symbolic state from the top of ST and store it in S. For each symbolic state S we will store the edge by which it was reached, and two integers, $S.\mathsf{index}$ and $S.\mathsf{lowlink}$. The integer variable $\mathsf{index} \in \mathbb{Z}$ is used to associate a unique integer value to each symbolic state; variable $\mathsf{lowlink} \in \mathbb{Z}$ holds the lowest value of index of the state that we can reach from the current state. $\mathsf{lowlink}$ and index have direct equivalents in Tarjan's algorithm [19].

The algorithm takes as input initial symbolic state S_0 and the symbolic goal state Goal. The algorithm will return **true**, if $\mathsf{Goal} \subseteq \mathcal{S}^*$ and **false** otherwise. The main algorithm is a depth-first search reachability algorithm with structures typical of Tarjan's algorithm [19]. The pseudo code is presented in Algorithm 1.

To find $\mathcal{S}^*$, the algorithm needs to find all stable zones that touch the set of reachable states. Unfortunately, we cannot calculate the stable zones by an a priori analysis of the edge cycles of the timed automaton as there are potentially infinitely many of them. Calculating stable zone W_σ only for the case where σ is a simple cycle (a cycle with no repeated nodes or edges) is not enough: simple cycles may have empty stable zones whereas combinations of them may have non-empty stable zones.

The approach we take is to do a depth first search exploration of the automaton and, for locations that we visit more than once, we calculate the set of stable zones (function `AllSZones`). Calculating `AllSZones`, for location q, is expensive if we do it for the whole automaton. To speed things up, we exclude locations from which we cannot go back to the starting location q. We do it by grouping locations into $\mathsf{SCS}_{\mathcal{U}}$ and limit `AllSZones` to one $\mathsf{SCC}_{\mathcal{U}}$ at a time.

The way we calculate all sets $\mathsf{SCS}_{\mathcal{U}}$ is similar to Tarjan's algorithm. The integer $S.\mathsf{index}$ numbers symbolic states in the order in which they were explored

Algorithm 1: DFS Reachability Algorithm based on Tarjan's algorithm

```
function Reach(S_0, Goal : Q × Z): B
    U := ∅; index := 1; ST := ∅; Q_mark := ∅                    // Initialisation
    return Search (↑S_0,Goal)                                   // Calling main function
function Search(S, Goal : Q × Z): B
    if Goal ⊆ S: return true
    else:
        U := U ∪ {S}                                            // Mark S as visited
        ST.push(S)                                              // Push S on the stack
        S.index := index                                        // Set the depth index for S
        S.lowlink := index                                      // Initialise lowlink for S
        index := index + 1
        foreach S', e: S ⇒^e ⇒^↑ S' do                          // Consider successors of S
            if S' ∈ U:
                if ∃S'' ∈ ST such that S' ⊆ S'':                // Is it backedge?
                    S.lowlink := min(S.lowlink, S''.lowlink)
                    q := location of S
                    Q_mark := Q_mark ∪ {q}                      // Mark q for stable zones
            else:                                               // S' has not been encountered
                if Search (↑S',Goal): return true
                S.lowlink := min(S.lowlink, S'.lowlink)
        if S.lowlink == S.index:                                // Is it a root node?
            SCS_U := ∅                                          // Set of locations SCS_U
            repeat                                              // Construct SCS_U
                S' := ST.pop()
                q := location of S'
                SCS_U := SCS_U ∪ {q}
            until S'==S
            foreach q ∈ Q_mark ∩ SCS_U do
                U' := AllSZones (SCS_U, q)                      // Get stable zones for SCS_U
                foreach S' ∈ U' do
                    if S ∩ S' ≠ ∅ and S' ∉ U:
                        if Search(↑S',Goal): return true
        return false
```

and lowlink is equal to index initially. It is updated to be the lowest index of states reachable from the given state. Each newly explored state is put on the stack ST. We remove states from ST only when we finished exploring a given state's successors and its lowlink equals its index. We also maintain a list of Q_{mark} locations. A location is put into Q_{mark} when there may be a potential stable zone passing through the location. This will be the case when the state is contained in some other state that we have seen before. After Q_{mark} and $\mathsf{SCS}_{\mathcal{U}}$ are created, we call `AllSZones`.

Algorithm 2: Function `AllSZones`($\mathsf{SCS}_{\mathcal{U}} : 2^{Q}$, $q : Q$): $2^{Q \times \mathcal{Z}}$

$\mathcal{U}_{\text{prev}} := \emptyset$; $\mathcal{U}_{\text{curr}} := \{(q, Z_\infty)\}$
while $\mathcal{U}_{\text{prev}} \neq \mathcal{U}_{\text{curr}}$**:**
  $\mathcal{U}_{\text{prev}} := \mathcal{U}_{\text{curr}}$
  $\mathcal{U}_{\text{curr}} := \emptyset$; $\mathsf{Open} := \emptyset$; $\mathcal{U}_{\text{seen}} := \emptyset$
  foreach $S \in \mathcal{U}_{\text{prev}}$ *s.t.* $e \vDash \mathsf{SCS}_{\mathcal{U}}$: *and* $S \xrightarrow{e} S'$ *and* $S' \notin \mathcal{U}$ **do**
    $\mathsf{Open.push}(\uparrow S')$
  while $\mathsf{Open} \neq \emptyset$**:**
    $S' := \mathsf{Open.pop}()$
    if $S'.\mathsf{location} == q$**:**
      $\sigma :=$ edge cycle by which we arrived to S' from location q
      if $\forall S \in \mathcal{U}_{\text{curr}} : pre_\sigma(S') \not\subseteq S$**:**
        $\mathcal{U}_{\text{curr}} := \mathcal{U}_{\text{curr}} \cup \{pre_\sigma(S')\}$
    $\mathcal{U}_{\text{seen}} := \mathcal{U}_{\text{seen}} \cup \{S'\}$
    foreach $S' \xrightarrow{e} S''$*:* $e \vDash \mathsf{SCS}_{\mathcal{U}}$ **do**
      if $S'' \notin \mathcal{U} \cup \mathcal{U}_{\text{seen}}$**:**
        $\mathsf{Open.push}(\uparrow S'')$
return $\mathcal{U}_{prev}$

3.2 Calculation of Stable Zones

The function `AllSZones`, called with arguments $(\mathsf{SCS}_{\mathcal{U}}, q)$, calculates a set of stable zones passing through a given location q. `AllSZones` is a fixpoint calculation of states in location q. Intuitively, fixpoint calculation is a result of Lemma 2. The fixpoint calculation uses two sets of symbolic states: $\mathcal{U}_{\text{prev}}$ and $\mathcal{U}_{\text{curr}}$ to store states at location q. $\mathcal{U}_{\text{curr}}$ is initialised with (q, Z_∞), where Z_∞ is a zone containing all clock valuations, that is a zone with an empty set of constraints. We maintain the set of states that we visited in this iteration step in the set $\mathcal{U}_{\text{seen}}$. The stack Open is similar to the stack ST. It holds the states that have their successors processed. The calculation is finished when $\mathcal{U}_{\text{prev}} = \mathcal{U}_{\text{curr}}$. In each iteration step we calculate the set of reachable states from $\mathcal{U}_{\text{prev}}$. In $\mathcal{U}_{\text{curr}}$ we store newly reached states for location q. We limit the generation of successors to the locations from $\mathsf{SCS}_{\mathcal{U}}$, that is, we consider edge e only if $e \vDash \mathsf{SCS}_{\mathcal{U}}$. When $\mathcal{U}_{\text{prev}} = \mathcal{U}_{\text{curr}}$, for each $S \in \mathcal{U}_{\text{curr}}$, we calculate the *pre* step and add the resulting state to the Open list.

3.3 Complexity

Checking reachability of a state in a timed automaton using semantics $[A]$ is a PSPACE-complete problem [2]. Let P be the time needed for checking reachability of a state. We will analyse the complexity of our algorithm relative to P. Let n be the number of clocks, k be the highest constant appearing in the specification, and m the number of locations in timed automaton A. In the pessimistic case, we may have to call `AllSZones` for each location and with $\mathsf{SCS}_{\mathcal{U}}$ containing all locations. To calculate `AllSZones` we may need k cycles, as a

stable zone may shrink only by one time unit for a computation cycle. A small improvement can be achieved thanks to the result presented in [16] in Lemma 2 on page 10. It states that if the fixpoint calculation of a stable zone has not finished after n^2 cycles then the stable zone is empty. Thanks to that we can limit the number of iterations in a fixpoint calculation to n^2. Thus, the worst case scenario complexity is $O(Pn^2m)$.

4 Correctness of the Algorithm

In this section we prove that the algorithm is correctly calculating the set of reachable states $\mathcal{S}^*$. The following lemma shows that by partitioning the locations into $\mathsf{SCS}_\mathcal{U}$, we will not omit any stable zones during calculations.

Lemma 3. *Let $\mathcal{U} \subseteq \mathcal{S}$ be a set of reachable states in $[A]$, and W_σ be a stable zone, $\sigma = e_1 \dots e_n$, and $W_\sigma \cap \mathcal{U} \neq \emptyset$. If $e_i \vDash \mathsf{SCS}_\mathcal{U}$ for some $1 \leq i \leq n$ then $\forall_{1 \leq j \leq n}\, e_j \vDash \mathsf{SCS}_\mathcal{U}$.*

Proof. Let q_i be the source location of e_i. Let $(q_i, v) \in W_\sigma \cap \mathcal{U}$ and $q_i \in \mathsf{SCS}_\mathcal{U}$. From Definition 7 it follows that there exists (q_i, v') such that $(q_i, v') \in W_\sigma \cap \mathcal{U}$ and $(q_i, v) \xrightarrow{\sigma} (q_i, v')$. Then the lemma follows directly from Definition 8.

The following theorem shows that if we call `AllSZones` $(\mathsf{SCS}_\mathcal{U}, q)$ then all stable zones passing through location q will be included in the result.

Theorem 1. *Let R be the result of* `AllSZones`*, called with input $(\mathsf{SCS}_\mathcal{U}, q)$, as presented in Algorithm 2. Then for all stable zones W_σ passing through location q, there exists $S \in R$ such that $W_\sigma \subseteq S$ or W_σ is reachable from S.*

Proof. We will prove this by induction on the number of times the while loop has been executed. Let $\mathcal{U}_i$ denote $\mathcal{U}_{\text{prev}}$ after the ith iteration of the while loop. Initially $\mathcal{U}_0 = (q, Z_\infty)$, so it trivially contains any possible stable zone passing through location q. Now let us assume that there exists $S \in \mathcal{U}_i$ such that $W_\sigma \subseteq S$ or W_σ is reachable from S. Inside a while loop we explore all states having locations from $\mathsf{SCS}_\mathcal{U}$. From Lemma 3, we know that we will be able to follow σ or any other edge cycle. From the properties of stable zones, we know that there exists $\mathcal{U}'$ such that $W_\sigma \in \mathcal{U}'$ and $\mathcal{U}'$ will be added to the Open stack. If $\mathcal{U}'$ was reached using σ then $\mathcal{U}_{i+1}$ will contain $\text{pre}_\sigma(W_\sigma) = W_\sigma$. If $\mathcal{U}'$ was reached using some other path $\sigma' \vDash \mathsf{SCS}_\mathcal{U}$, then we know that $\text{pre}_{\sigma'}(W_\sigma) \cap W_\sigma$ will be added to $\mathcal{U}_{i+1}$ and we know that we can reach W_σ from $\mathcal{U}_{i+1}$ using σ' which concludes the inductive proof.

The following theorem shows that the sequence of sets $\mathcal{U}_1, \dots, \mathcal{U}_i$ calculated in the function `AllSZones` is non-increasing. This shows that the function `AllSZones` terminates.

Theorem 2. *Let $\mathcal{U}_i$ denote $\mathcal{U}_{prev}$ after the ith iteration of the while loop in the function* `AllSZones` *$(\mathsf{SCS}_\mathcal{U}, q)$ presented in Algorithm 2. Then $\forall_{i>0} \forall S \in \mathcal{U}_{i+1}\ \exists S' \in \mathcal{U}_i$ such that $S \subseteq S'$.*

Proof. In order to reduce the number of quantifiers, we will use $s \in \mathcal{U}$ to denote $\exists S \in \mathcal{U} : s \in S$.

The proof will proceed by induction over the loop number i. Initially $\mathcal{U}_1 = (q, Z_\infty)$, so trivially all elements of $\mathcal{U}_2$ are included in $\mathcal{U}_1$. Let us assume that $\forall_{1 \leq k < i}\ \forall S \in \mathcal{U}_{k+1}\ \exists S' \in \mathcal{U}_k$ such that $S \subseteq S'$. We need to show that this property holds for $k = i$.

Because all elements of $\mathcal{U}_{i+1}$ are reachable from $\mathcal{U}_1$, $\forall (q, Z_{i+1}) \in \mathcal{U}_{i+1}$ there exists a trajectory $(q, Z_1) \rightarrow^* (q, Z_2) \rightarrow^* \ldots \rightarrow (q, Z_i) \rightarrow^* (q, Z_{i+1})$ such that $\forall_{1 \leq j \leq i+1}\ (q, Z_j) \in \mathcal{U}_j$. Using the induction assumption, we know that $(q, Z_i) \in \mathcal{U}_{i-1}$. We explore all possible paths in $\mathsf{SCS}_{\mathcal{U}}$ from $\mathcal{U}_{i-1}$ to calculate $\mathcal{U}_i$. Thus, if $(q, Z_i) \in \mathcal{U}_{i-1}$, and $(q, Z_i) \xrightarrow{\sigma} (q, Z_{i+1})$, and $\sigma \vDash \mathsf{SCS}_{\mathcal{U}}$ then $(q, Z_{i+1}) \in \mathcal{U}_i$, which finishes the proof.

The next theorem states that all relevant zones are added to the set of reachable states $\mathcal{U}$ by Algorithm 1. Together with Lemma 1 this proves completeness and safety of Algorithm 1.

Theorem 3. *Given timed automaton A, let $\mathcal{U}$ be a set of reachable states in standard semantics. Then all stable zones W_σ such that $W_\sigma \cap \mathcal{U} \neq \emptyset$ are included in the result of Algorithm 1.*

Proof. For any given zone W_σ, let q be the first location from σ such that part of W_σ is reached when doing reachability analysis. From the properties of stable zones, we know that location q will be visited at least once more. When location q is visited for the last time, the current state will be contained in the $\mathcal{U}$ list. Thus, location q will be added to Q_{mark}. As a result, function `AllSZones` will be called for location q and using Theorem 1, we know that the zone W_σ will be added to the reachable set of states or it will be discovered during later exploration. Thus the zone W_σ will be included in the result of Algorithm 1.

5 Implementation and Experiments

5.1 Implementation

To prove that our algorithm is applicable in practice, we have implemented a prototype tool and tested it on a number of examples. The tool can perform the reachability analysis in standard semantics using a breadth-first or depth-first search and in extended semantics using Algorithm 1. The tool has been written in C++ and the source code has about 5.6 Klocs. Internally it uses the Uppaal DBM Library[3] and Uppaal Timed Automata Parser Library[4]. The input format for the tool is in the XML format compatible with Uppaal. The tool, source, and specifications used for experiments can be downloaded from `http://satoss.uni.lu/members/piotr/verifix/`.

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

5.2 Experiments

To check the performance of our implementation, we have used the examples from the suite of Uppaal benchmarks taken from the Uppaal web-page [17].

As the first benchmark, we have used the CSMA/CD protocol (Carrier Sense, Multiple-Access with Collision Detection). This is a media access control protocol used most notably in local area networking with early Ethernet technology. A detailed description of the protocol can be found in [22]. We have verified the protocol with nine, ten, and eleven components.

Another example is Fisher's Protocol which is a mutual exclusion algorithm, described in [1]. We have verified the protocol for eight and nine components.

The FDDI (Fiber Distributed Data Interface) is a fiber-optic token ring local area network, described e.g. in [15, 9]. FDDI networks are composed of N symmetric stations that are organized in a ring. We have used a simplified model with 11-ary, 12-ary, and 13-ary networks.

The next verified model is a Mutual Exclusion protocol. The protocol ensures mutual exclusion of a state in a distributed system via asynchronous communication. The protocol is described in full detail in [12]. For verification we used models with three and four components.

The last model we have used is a lip synchronisation protocol described in [6]. Specifically we have used the model which assumes an ideal video stream. We simplified the model by scaling down each constant by a factor of ten and five, as the original model proved to be too difficult to verify within reasonable time.

Run type / Model	Breadth-First		Depth-First			Extended Sem.		
	run time	gen. states	run time	gen. states	err. loc.	run time	gen. states	err. loc.
CSMA/CD – 9 comp.	0.3s	110k	0.7s	231k	no	0.9s	260k	no
CSMA/CD – 10 comp.	1.0s	287k	2.7s	775k	no	3.5s	846k	no
CSMA/CD – 11 comp.	2.9s	728k	11.6s	2 607k	no	15.1s	2 790k	no
Fisher – 8 comp.	0.3s	180k	0.4s	222k	no	263.9s	348k	no
Fisher – 9 comp.	1.8s	723k	2.4s	1 004k	no	6151s	1 447k	no
FDDI – 11-ary network	1.7s	28k	8.4s	68 k	no	62.1s	122k	no
FDDI – 12-ary network	10.4s	55k	42.3s	137k	no	263.7s	224k	no
FDDI – 13-ary network	53.4s	109k	187.5s	278k	no	905.5s	1 775k	no
Mutual excl. – 3 comp.	0.1s	61k	0.1s	35k	no	7.5s	1 047k	no
Mutual excl. – 4 comp.	4.8s	1 506k	2.2s	866k	no	3545s	28 272k	no
Lip synchr. – scale 0.1	0.06s	1 22k	0.06s	22k	no	30.1s	5 244k	yes
Lip synchr. – scale 0.2	0.1s	1 22k	0.1s	22k	no	1537s	67 222k	yes

Table 1. Validation time in seconds and number of generated states in thousands.

The experiments have been performed on an Intel i7 processor with a 2.4 GHz system and 8 GByte of memory. The results for all the verified models were the same for standard semantics – the models are correct and the error location is

not reached. In extended semantics only the Lip Synchonization Protocol proved to be non-robust as we managed to reach the error location – meaning that video and sound can desynchronize over time). The running times and number of generated states for each verification model are shown in Table 1.

The results show that for the CSMA/CD protocol our algorithm is almost as good as depth first search. The performance drops drastically when we have a model with many parallel components like Fisher's protocol and the Mutual Exclusion protocol. For those two cases $\mathsf{SCS}_{\mathcal{U}}$ will contain almost all locations of timed automaton A, implying that the function `AllSZones` must do reachability on the original A, which is the main source of inefficiency. Checking reachability in the proposed robust way can be up to thousand times slower. The inefficiency comes from the fact that we define $\mathsf{SCS}_{\mathcal{U}}$ in terms of locations. Potentially, this may lead to many calls to the `AllSZones` function for the same input. The solution could be to introduce some form of caching or redefine $\mathsf{SCS}_{\mathcal{U}}$ to include the timing aspect. The verification of the Lip Synchronization Protocol proved to be the hardest. The specification uses a discrete variable as a discrete clock which creates many stable zones that are not reachable from each other in standard semantics, but are touching each other. As a result they are reachable in extended semantics, but the algorithm needs to add new zones many times.

6 Conclusions and Perspectives

Our research shows the feasibility of automated verification of systems with the (realistic) assumption of drifting clocks. Many communication protocols and distributed algorithms have already been designed with this assumption in mind, but up to now they could not be formally verified.

The main result of our work is the development of a symbolic algorithm for computing the reachability of locations in timed automata when the clocks may drift by an arbitrarily small amount. The analysis on which this algorithm is based was originally defined [18, 11] for region automata and does not lend itself to direct implementation in a tool. Our zone-based implementation is therefore a very important step towards the analysis of robustness of timed systems. The key concept on which our analysis is based is the notion of a stable zone, originally defined in [8]. Unfortunately the concept of stable zone can be used only for analysis of flat timed automata, which seriously limits this approach. Our zone-based approach does not have this limitation.

We have developed a tool written in C++ implementing our algorithm. The tool was tested on a number of benchmark specifications. The tests have shown that the extended semantics performs three to four times slower in most cases, but for some highly parallel specifications the verification time can be up to thousand times slower, especially for specifications that are not robust.

In the future, we are interested in removing the limitations of bounded clocks and the necessity of progress cycles. While removing the limitations may be rather straight-forward to solve for flat automata, it is non-trivial in the general case. If the extension is successful it would be interesting to check the robustness

of a wider class of real-life specifications. It will be interesting to investigate a priori conditions for robustness, and techniques to repair robustness violations.

References

1. Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer, 2003.
4. Patricia Bouyer, Nicolas Markey, and Pierre-Alain Reynier. Robust model-checking of linear-time properties in timed automata. In *LATIN*, pages 238–249, 2006.
5. Patricia Bouyer, Nicolas Markey, and Pierre-Alain Reynier. Robust analysis of timed automata via channel machines. In *FoSSaCS*, pages 157–171, 2008.
6. H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip-synchronisation protocol using uppaal. *Formal Aspects of Computing*, 10(5-6):550–575, 1998.
7. Hubert Comon and Yan Jurski. Timed automata and the theory of real numbers. In *CONCUR'99 Concurrency Theory*, LNCS, pages 242–257. Springer, 1999.
8. Conrado Daws and Piotr Kordy. Symbolic robustness analysis of timed automata. In *FORMATS*, LNCS, pages 143–155. Springer, 2006.
9. Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, pages 313–329, 1998.
10. Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robustness and implementability of timed automata. In *FORMATS/FTRTFT*, volume 3253 of *LNCS*, pages 118–133. Springer, 2004.
11. Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-François Raskin. Robust safety of timed automata. *Formal Meth. Syst. Des.*, 33(1-3):45–84, 2008.
12. Henning Dierks. Comparing model checking and logical reasoning for real-time systems. *Formal Asp. Comput.*, 16(2):104–120, 2004.
13. Cătălin Dima. Dynamical properties of timed automata revisited. In *Formal Modeling and Analysis of Timed Systems*, LNCS, pages 130–146. Springer, 2007.
14. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.
15. Raj Jain. *FDDI Handbook: High-Speed Networking Using Fiber and Other Media.* Addison Wesley Publishing Company, 1994.
16. Rémi Jaubert and Pierre-Alain Reynier. Quantitative robustness analysis of flat timed automata. In *FoSSaCS'11*, LNCS, pages 229–244. Springer, 2011.
17. Department of Information Technology at Uppsala University and the Department of Computer Science at Aalborg University. UPPAAL. `http://www.uppaal.org/`.
18. Anuj Puri. Dynamical properties of timed automata. *Discrete Event Dynamic Systems-Theory and Applications*, 10(1-2):87–113, 2000.
19. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
20. Howard Wong-Toi. Analysis of slope-parametric rectangular automata. In *Hybrid Systems V*, pages 390–413, London, UK, 1999. Springer.
21. Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. In *CAV*, LNCS, pages 210–224. Springer, 1993.
22. Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.