Lazy Abstractions for Timed Automata*

Frédéric Herbreteau[1], B. Srivathsan[2], and Igor Walukiewicz[1]

[1] Univ. Bordeaux, CNRS, LaBRI, UMR 5800, F-33400 Talence, France
[2] Software Modeling and Verification Group, RWTH Aachen University, Germany

Abstract. We consider the reachability problem for timed automata. A standard solution to this problem involves computing a search tree whose nodes are abstractions of zones. For efficiency reasons, they are parametrized by the maximal lower and upper bounds (LU-bounds) occurring in the guards of the automaton. We propose an algorithm that dynamically updates LU-bounds during exploration of the search tree. In order to keep them as small as possible, the bounds are refined only when they enable a transition that is impossible in the unabstracted system. So our algorithm can be seen as a kind of lazy CEGAR algorithm for timed automata. We show that on several standard benchmarks, the algorithm is capable of keeping very small LU-bounds, and in consequence is able to reduce the search space substantially.

1 Introduction

Timed automata are obtained from finite automata by adding clocks that can be reset and whose values can be compared with constants. The reachability problem asks if a given target state is reachable from the initial state by an execution of the automaton. The standard solution to this problem computes the so-called zone graph of the automaton, and uses abstractions to make the algorithm both terminating and more efficient.

Most abstractions are based on constants used in comparisons of clock values. Such abstractions have already been considered in the seminal paper of Alur and Dill [1]. Behrmann et. al. [4] have proposed abstractions based on LU-bounds, that are two functions L and U: the L function assigns to every clock a maximal constant appearing in a lower bound constraint in the automaton; similarly U associates the maximum constant appearing in an upper bound constraint. In a recent paper [15] we have shown how to efficiently use the $\mathfrak{a}_{\preccurlyeq LU}$ abstraction of [4] that is parameterized by LU-bounds. Moreover, $\mathfrak{a}_{\preccurlyeq LU}$ has been proved to be the biggest abstraction that is sound for all automata with given LU-bounds. Since $\mathfrak{a}_{\preccurlyeq LU}$ abstraction of a zone can result in a non-convex set, we have shown in [15] how to use this abstraction without the need to store the result of the abstraction. This opens new algorithmic possibilities because changing LU-bounds becomes very cheap as abstractions need not be recalculated. In this paper we explore these possibilities.

* This work was partially supported by the ANR project Vacsim (ANR-11-INSE-004).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 990–1005, 2013.

The algorithm we propose works as follows. It constructs a graph with nodes of the form (q, Z, LU), where q is a state of the automaton, Z is a zone, and LU are parameters for the abstraction. It starts with the biggest abstraction: LU bounds are set to $-\infty$ which makes $\mathfrak{a}_{\preccurlyeq LU}(Z)$ to be the set of all valuations for every nonempty Z. The algorithm explores the zone graph using standard transition relation on zones, without modifying LU bounds till it encounters a disabled transition. More concretely, till it reaches a node (q, Z, LU) such that there is a transition from q that is not possible from (q, Z) because no valuation in Z allows to take it. At this point we need to adjust LU bounds so that the transition is not possible from $\mathfrak{a}_{\preccurlyeq LU}(Z)$ either. This adjustment is then propagated backwards through the already constructed part of the graph.

The real challenge is to limit the propagation of bound updates. For this, if the bounds have changed in a node $(q', Z', L'U')$ then we consider its predecessor nodes (q, Z, LU) and update its LU bounds as a function of Z, Z' and $L'U'$. We give general conditions for correctness of such an update, and a concrete efficient algorithm implementing it. This requires getting into a careful analysis of the influence of the transition on the zone Z. As a result we obtain an algorithm that exhibits exponential gains on some standard benchmarks.

We have analyzed the performance of our algorithm theoretically as well as empirically. We have compared it with a static analysis algorithm, namely the state-of-the-art algorithm implemented in UPPAAL, and with an algorithm we have proposed in [14]. The latter improves on the static analysis algorithm by considering only the reachable part of the zone graph. For an example borrowed from [17] we have proved that the algorithm presented here produces a linear size search graph while for the other two algorithms, the search graph is exponential in the size of the model. For the classic FDDI benchmark, which has been tested on just about every algorithm for the reachability problem, our algorithm shows the rather surprising fact that the time component is almost irrelevant. There is only one constraint that induces LU bounds, and in consequence the abstract search graph constructed by our algorithm is linear in the size of the parameter of FDDI.

Our algorithm can be seen as a kind of CEGAR algorithm similar in spirit to [13], but then there are also major differences. In the particular setting of timed automata the information available is much richer, and we need to use it in order to obtain a competitive algorithm. First, we do not need to wait till a whole path is constructed to analyze if it is spurious or not. Once we decide to keep zones in nodes we can immediately detect if an abstraction is too large: it is when it permits a transition not permitted from the zone itself. Next, the abstractions we use are highly specialized for the reachability problem. Finally, the propagation of bound changes gets quite sophisticated because it can profit from the large amount of useful information in the exploration graph.

Related work. Forward analysis is the main approach for the reachability testing of real-time systems. The use of zone-based abstractions for termination has been introduced in [10]. The notion of LU-bounds and inference of these bounds by static analysis of an automaton have been proposed in [3,4]. The $\mathfrak{a}_{\preccurlyeq LU}$

approximation has been introduced in [4]. An approximation method based on LU-bounds, called $Extra^+_{LU}$, is used in the current implementation of UPPAAL [5]. In [15] we have shown how to efficiently use $\mathfrak{a}_{\preccurlyeq_{LU}}$ approximation. We have also proposed an LU-propagation algorithm [14] that can be seen as applying the static analysis from [3] on the zone graph instead of the graph of the automaton; moreover this inference is done on-the-fly during construction of the zone graph. In the present paper we do much finer inference and propagation of LU-bounds.

Approximation schemes for the analysis of timed-automata have been considered almost immediately after the introduction of the concept of timed automata, as for example in [2,22,12] or [20]. All these papers share the same idea to abstract the region graph by not considering all the constraints involved in the definition of a region. When a spurious counterexample is discovered a new constraint is added. So in the worst case the whole region graph will be constructed. Our algorithm in the worst case constructs an $\mathfrak{a}_{\preccurlyeq_{LU}}$-abstracted zone graph with LU-bounds obtained by static analysis. This is as good as the state-of-the-art method used in UPPAAL. Another slightly related paper is [7] where the CEGAR approach is used to handle diagonal constraints.

Let us mention that abstractions are not needed in the backward exploration of timed systems. Nevertheless, any feasible backward analysis approach needs to simplify constraints. For example [19] does not use approximations and relies on an SMT solver instead. This approach, or the approach of RED [21], are very difficult to compare with the forward analysis approach we study here.

Organization of the paper. In the preliminaries section we introduce all standard notions we will need, and $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction in particular. Section 3 gives a definition of adaptive simulation graph (ASG). Such a graph represents the search space of a forward reachability testing algorithm that will search for an abstract run with respect to $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction, while changing LU-bounds dynamically during exploration. Section 4 gives an algorithm for constructing an ASG with small LU-bounds. Section 5 presents the two crucial functions used in the algorithm: the one updating the bounds due to disabled edges, and the one propagating the change of bounds. Section 6 explains some advantages of our algorithm on variations of an example borrowed from [17]. The experiments section compares our prototype tool with UPPAAL, and our algorithm from [14]. The conclusions section gives some justification for our choice of concentrating on LU-bounds. The proofs are presented in the full version of the paper [16].

2 Preliminaries

2.1 Timed Automata, Zones, and the Reachability Problem

Let X be a set of clocks, i.e., variables that range over $\mathbb{R}_{\geq 0}$, the set of non-negative real numbers. A *guard* is a conjunction of constraints $x\#c$ for $x \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$, e.g. $(x \leq 3 \wedge y > 0)$. Let $\Phi(X)$ denote the set of clock constraints over clock variables X. A *clock valuation* over X is a function

$v : X \to \mathbb{R}_{\geq 0}$. We denote by $\mathbf{0}$ the valuation that associates 0 to every clock in X. We write $v \vDash \phi$ when v satisfies the guard ϕ. For $\delta \in \mathbb{R}_{\geq 0}$, let $v + \delta$ be the valuation that associates $v(x) + \delta$ to every clock x. For $R \subseteq X$, let $v[R]$ be the valuation that sets x to 0 if $x \in R$, and that sets x to $v(x)$ otherwise.

A *timed automaton (TA)* is a tuple $\mathcal{A} = (Q, q_0, X, T, Acc)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, X is a finite set of clocks, $Acc \subseteq Q$ is a set of accepting states, and $T \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a finite set of transitions (q, g, R, q') where g is a *guard*, and R is the set of clocks that are *reset* on the transition.

A *configuration* of $\mathcal{A}$ is a pair $(q, v) \in Q \times \mathbb{R}_{\geq 0}^X$ and $(q_0, \mathbf{0})$ is the *initial configuration*. We have two kinds of transitions:

Delay: $(q, v) \to^{\delta} (q, v + \delta)$ for some $\delta \in \mathbb{R}_{\geq 0}$;

Action: $(q, v) \to^{t} (q, v[R])$ for a transition $t = (q, g, R, q') \in T$ such that $v \vDash g$.

A *run* of $\mathcal{A}$ is a finite sequence of transitions starting from the initial configuration $(q_0, \mathbf{0})$. A run is *accepting* if it ends in a configuration (q_n, v_n) with $q_n \in Acc$.

Definition 1 (Reachability problem). *The* reachability problem *for timed automata is to decide whether there exists an accepting run of a given automaton.*

This problem is known to be PSPACE-complete [1,9]. The class of TA we consider is usually known as diagonal-free TA since clock comparisons like $x - y \leq 1$ are disallowed. Notice that if we are interested in state reachability, considering timed automata without state invariants does not entail any loss of generality as the invariants can be added to the guards. For state reachability, we can also consider automata without transition labels.

Rather than working with valuations we will work with sets of valuations and symbolic transitions. So we will consider configurations of the form (q, W) where q is a state of the automaton and W a set of valuations.

Definition 2 (Symbolic transition $\Rightarrow$). *Let $\mathcal{A}$ be a timed automaton. For every transition t of $\mathcal{A}$ and every set of valuations W, we define:*

$$(q, W) \Rightarrow^{t} (q', W') \text{ where } W' = \{v' \mid \exists v \in W,\ \exists \delta \in \mathbb{R}_{\geq 0}.\ (q, v) \to^{t} \to^{\delta} (q', v')\}$$

We will sometimes write $\mathrm{Post}_t(W)$ *for W'. The transition relation $\Rightarrow$ is the union of all $\Rightarrow^t$.*

Observe that symbolic transitions always yield sets closed under time-successors. Such sets are called *time-elapsed*. Let W_0 be the set of valuations reachable from the the initial valuation $\mathbf{0}$ by a time elapse: $W_0 = \{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$.

A *symbolic run* is a sequence of symbolic transitions:

$$(q_0, W_0) \Rightarrow (q_1, W_1) \Rightarrow \dots$$

It has been noticed that the sets W appearing in a symbolic run can be described by some simple constraints involving only the difference between clocks [6]. This has motivated the definition of *zones*, which are sets of valuations defined by difference constraints.

Definition 3 (Zones [6]). *A zone is a set of valuations defined by a conjunction of two kinds of clock constraints: $x \sim c$ and $x - y \sim c$ for $x, y \in X$, $c \in \mathbb{Z}$, and $\sim \in \{\leq, <, =, >, \geq\}$.*

Observe that W_0 is a zone: it is given by the constraints $\bigwedge_{x,y \in X} (x \geq 0 \wedge x - y = 0)$. Then every W appearing in a symbolic run is a zone too. It is well-known that an automaton has an accepting run if and only if it has a symbolic run reaching an accepting state and non-empty zone. As the number of zones that can appear in a symbolic run is not bounded, the next simplification step is to consider abstractions of zones.

2.2 LU-bounds and LU-abstractions

The most common parameter taken for defining abstractions are LU-bounds.

Definition 4 (LU-bounds). *The L bound for an automaton $\mathcal{A}$ is the function assigning to every clock x a maximal constant that appears in a lower bound guard for x in $\mathcal{A}$, that is, the maximum over guards of the form $x > c$ or $x \geq c$. Similarly U is the function assigning to every clock x a maximal constant appearing in an upper bound guard for x in $\mathcal{A}$, that is, the maximum over guards of the form $x < c$ or $x \leq c$. If there is no guard in question then the value of $L(x)$ or $U(x)$ is $-\infty$.*

The paper introducing LU-bounds [4] also introduced an abstraction operator $\mathfrak{a}_{\preccurlyeq_{LU}}$ that uses LU-bounds as parameters. We begin by recalling the definition of an LU-preorder defined in [4]. We use a different but equivalent formulation.

Definition 5 (LU-preorder and $\mathfrak{a}_{\preccurlyeq_{LU}}$-abstraction [4]). *Let $L, U : X \to \mathbb{N} \cup \{-\infty\}$ be two bound functions. For a pair of valuations we set $v \preccurlyeq_{LU} v'$ if for every clock x:*

- *if $v'(x) < v(x)$ then $v'(x) > L_x$, and*
- *if $v'(x) > v(x)$ then $v(x) > U_x$.*

For a set of valuations W we define:

$$\mathfrak{a}_{\preccurlyeq_{LU}}(W) = \{v \mid \exists v' \in W.\ v \preccurlyeq_{LU} v'\}.$$

An efficient algorithm to use the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction for reachability was proposed in [15]. Moreover in op. cit. it was shown that over time-elapsed zones, $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction is optimal when the only information about the analyzed automaton are its LU-bounds. Informally speaking, for a fixed LU, the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction is the biggest abstraction that is sound and complete for all automata using guards within LU-bounds.

Since the abstraction $\mathfrak{a}_{\preccurlyeq_{LU}}$ is optimal, the next improvement is to try to get as good LU-bounds as possible since tighter bounds give coarser abstractions, and in consequence induce a smaller search space.

It has been proposed in [3] that instead of considering one LU-bound for all states in an automaton, one can use different bound functions for each state. For

every state q and every clock x, constants $L_x(q)$ and $U_x(q)$ are determined by the least solution of the following set of inequalities. For each transition (q, g, R, q') in the automaton, we have:

$$\begin{cases} L_x(q) \geq c & \text{if } x \geq c \text{ or } x > c \text{ is a constraint in } g \\ L_x(q) \geq L_x(q') & \text{if } x \notin R \end{cases} \tag{1}$$

Similar inequalities are written for U, now considering $x \leq c$ and $x < c$. It has been shown in [3] that such an assignment of constants is sound and complete for state reachability. Experimental results have shown that this method, that performs a static analysis on the automaton, often gives very big gains.

3 Adaptive Simulation Graph

In this paper we improve on the idea of static analysis that computes LU-bounds for each state q. We will compute LU-bounds on-the-fly for each node (q, Z) of the zone graph, while simultaneously searching for an accepting run. The key difference is that the bounds will depend not only on the state but also on the set of valuations. This will immediately allow us to ignore guards from unreachable parts of the automaton. However, the real freedom given by an adaptive simulation graph and Theorem 1 presented below is that when calculating the LU-bounds, they will allow to ignore some guards of transitions even from the reachable part. As we will see in the experiments section, this can result in significant gains.

We will construct a forward reachability algorithm that will search for an abstract run with respect to $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction, where the LU-bounds change dynamically during exploration. The intuition of a search space of such an algorithm is formalized in a notion of an adaptive simulation graph (ASG). Such a graph permits to change LU-bounds from node to node, provided some consistency conditions are satisfied. This is important as LU-bounds are used to stop exploring successors of a node. So our goal will be to find as small LU-bounds as possible in order to stop developing the graph as soon as possible.

Definition 6 (Adaptive simulation graph (ASG)). *Fix an automaton $\mathcal{A}$. An ASG has nodes of the form (q, Z, LU) where q is the state of $\mathcal{A}$, Z is a zone, and LU are bound functions. Some nodes are declared to be tentative. The graph is required to satisfy three conditions:*

G1 *For the initial state q_0 and initial zone Z_0, a node (q_0, Z_0, LU) should appear in the graph for some LU.*

G2 *If a node (q, Z, LU) is not tentative then for every transition $(q, Z) \Rightarrow_t (q', Z')$ the node should have a successor labeled $(q', Z', L'U')$ for some $L'U'$.*

G3 *If a node (q, Z, LU) is tentative then there should be a non-tentative node $(q', Z', L'U')$ such that $q = q'$ and $Z \subseteq \mathfrak{a}_{\preccurlyeq_{L'U'}}(Z')$. Node $(q', Z', L'U')$ is called covering node.*

We will also require that the following invariants are satisfied:

I1 *If a transition* $\Rightarrow^t$ *is disabled from* (q, Z)*, and* (q, Z, LU) *is a node of the ASG then* $\Rightarrow^t$ *should be disabled from* $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ *too;*

I2 $\mathrm{Post}_t(\mathfrak{a}_{\preccurlyeq_{LU}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq_{L'U'}}(Z')$*, for every edge* $(q, Z, LU) \Rightarrow_t (q', Z', L'U')$ *of the ASG.*

I3 $L_2U_2 \leq L_1U_1$*, for every tentative node* (q, Z_1, L_1U_1) *and the corresponding covering node* (q, Z_2, L_2U_2)*.*

In the above $L_2U_2 \leq L_1U_1$ says that for all clocks x, $L_2(x) \leq L_1(x)$ and $U_2(x) \leq U_1(x)$. The conditions G1, G2, G3 express the expected requirements for a graph to cover all reachable configurations. In particular, the condition G3 allows to stop the exploration if there is already a "better" node in the graph. The three invariants are more subtle. They imply that LU-bounds should be big enough for the reachability information to be preserved. (cf. Theorem 1).

Remark: While the idea is to work with nodes of the form (q, W) with W being a result of $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction, we do not want to store W directly, as we have no efficient way of representing and manipulating such potentially non-convex sets. Instead we represent each W as $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$. So we store Z and LU. This choice is algorithmically cheap since testing the inclusion $Z' \subseteq \mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ is practically as easy as testing $Z' \subseteq Z$ [15]. This approach has another big advantage: when we change the LU bound in a node, we do not need to recalculate $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$.

Remark: It is important to observe that for every $\mathcal{A}$ there exists a finite ASG. For this it is sufficient to take static LU-bounds as described by (1). It means that we can take the ASG whose nodes are $(q, Z, L(q)U(q))$ with bound functions given by static analysis [3]. It is easy to see that such a choice makes all three invariants hold.

The next theorem tells us that any ASG is good enough to determine the existence of an accepting run. Our objective in the following section will be to construct an ASG as small as possible.

Theorem 1. *Let* G *be an ASG for an automaton* $\mathcal{A}$*. An accepting state is reachable by a run of* $\mathcal{A}$ *iff a node containing an accepting state of* $\mathcal{A}$ *and a non-empty zone is reachable from the initial node of* G*.*

4 Algorithm

In this section, we present an algorithm that computes an ASG satisfying conditions G1, G2, G3 and maintaining invariants I1, I2, I3 of Definition 6. The basic algorithm is the same as in [14]. The LU-bounds are calculated dynamically while constructing the ASG. The main difference lies in the way LU-bounds are calculated so that the ASG is small, but still maintains the invariants.

The algorithm constructs the ASG in a form of a tree with cross edges from tentative nodes. The nodes v in this tree consist of four components: $v.q$ is a state of $\mathcal{A}$, $v.Z$ is a zone, and $v.L$, $v.U$ are LU bound functions. Each node v has a successor v_t for every transition t of $\mathcal{A}$ that results in a non-empty zone

from $v.Z$. Some nodes will be marked tentative and not explored further. After an exploration phase, tentative nodes will be reexamined and some of them will be put on the stack for further exploration. At every point the leaves of the tree constructed by the algorithm will be of three kinds: tentative nodes, nodes on the stack, nodes having no transition to be explored.

The exploration proceeds by a standard depth-first search. When a node v is called for exploration, we assume that the values $v.q$ and $v.Z$ are set. Moreover, $v.Z$ must be non-empty. The initial values of $v.L$ and $v.U$ are set to $-\infty$. We also assume that the constructed tree satisfies the invariants I1, I2, I3, except for the node v and the nodes on the stack. If the state $v.q$ is accepting then we have found an accepting run and the algorithm terminates reporting "not empty". Otherwise, the procedure needs to explore the successors of v and restore invariants, if needed. For this, it is first checked if there exists a non-tentative node v' in the tree such that $v.Z \subseteq \mathfrak{a}_{\preccurlyeq v'.LU}(v'.Z)$. If it is true, then v is marked tentative wrt v' and $v.LU$ is set to $v'.LU$ in order to maintain I3. The node v is not explored further. If such a non-tentative node cannot be found in the tree, the successors of v are computed and put on the stack. To ensure I1, a function `disabled` is called that gives new bounds for $v.LU$. We explain this function in the next section.

When LU-bounds in some node v are changed, the invariant I2 should be restored for its ancestors. For this, the modified bounds are propagated upward along the tree. The parent v_p of v is taken and the transition from v_p to v is examined. A function `newbounds` is called on v_p. This function calculates new LU bounds for a node given the changes in its successor, so that I2 is ensured. This function is the core of our algorithm and is the subject of the next section. If the bounds of v_p indeed change then they should be copied to all nodes tentative w.r.t. v_p. This is necessary to satisfy the invariant I3. Finally the bounds are propagated to the predecessor of v_p to restore invariant I2.

An exploration phase stops if there are no more nodes in the stack. During the course of the exploration, the LU bounds of tentative nodes might have changed. A procedure `resolve` is called to check for the consistency of tentative nodes. If v is tentative w.r.t. v' but $v.Z \not\subseteq \mathfrak{a}_{\preccurlyeq v'.LU}(v'.Z)$ is not true anymore, v needs to be explored. Hence it is viewed as a new node, marked non-tentative, and put on the stack for further exploration.

The algorithm terminates when either it finds an accepting state, or there are no nodes to be explored and all tentative nodes remain tentative. In the second case we can conclude that the constructed tree represents an ASG, and hence no accepting state is reachable. Note that the overall algorithm should terminate as the bounds can only increase and bounds in a node (q, Z) are not bigger than the bounds obtained for q by static analysis (cf. Remark on page 996). The correctness of the algorithm then follows from Proposition 1.

Proposition 1. *The algorithm always terminates. If for a given $\mathcal{A}$ the result is "not empty" then $\mathcal{A}$ has an accepting run. Otherwise the algorithm returns empty after constructing ASG for $\mathcal{A}$ and not seeing an accepting state.*

5 Controlling *LU*-bounds

In the previous section, we described the basic algorithm to construct an ASG with LU-bounds at each node. We had not addressed the issue of how to calculate these LU-bounds. In this section we describe the two functions used by the algorithm to calculate LU-bounds: `disabled` and `newbounds`.

The notion of adaptive simulation graph (Definition 6) gives necessary conditions for the values of LU bounds in every node. The invariant I1 tells that LU bounds in a node should take into account the edges disabled from the node. The invariant I2 gives a lower bound on LU with respect to the LU-bounds in successors of the node. Finally, I3 tells us that LU bounds in a covered node should not be smaller than in the covering node. The task of the functions `disabled` and `newbounds` is precisely to maintain the three invariants without increasing the bounds unnecessarily. The pseudo-code of these functions is presented in [16].

Proviso: For simplicity, we assume a special form of transitions of timed automata. A transition can have either only upper bound constraints, or only lower bound constraints and no resets. Observe that a transition $q_1 \xrightarrow{g;R} q_2$ is equivalent to $q_1 \xrightarrow{g_L} q_1' \xrightarrow{g_U;R} q_2$; where g_L is the conjunction of the lower bound guards from g and g_U is the conjunction of the upper bound guards from g. So in order to satisfy our proviso we may need to double the number of states of an automaton.

The `disabled` function is quite simple. Its task is to restore the invariant I1. For this the function chooses from every disabled transition an atomic guard that makes it disabled. Recall that we have assumed that every guard contains either only lower bound constraints or only upper bound constraints. A transition with only lower bound constraints cannot be disabled from a time elapsed zone. Hence a guard on a disabled transition must be a conjunction of upper bound constraints. It can be shown that if such a guard is not satisfied in a zone then there is one atomic constraint of the guard that is not satisfied in a zone. Now it suffices to use Definition 5 and observe that if a constraint $x \leq d$ or $x < d$ is not satisfied in Z then it is not satisfied in $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ when $U(x) \geq d$.

In the rest of this section we describe the function $\texttt{newbounds}(v, v', X'_L, X'_U)$. This function calculates new LU-bounds for v, given that the bounds in v' have changed. As an additional information we use the sets of clocks X'_L and X'_U that have changed their L-bound, and U-bound respectively, in v'. This information makes the `newbounds` function more efficient since the new bounds depend only on the clocks in X'_L and X'_U. The aim is to give bounds that are as small as possible and at the same time satisfy invariant I2 from Definition 6.

For space reasons we will consider only the case when a guard is a single constraint and there is no reset. The extension to a conjunction of constraints does not pose particular problems. Treating reset is easy. We treat transitions having guard and reset at the same time as a combination of purely guard and purely reset transitions. We refer the reader to the full paper for details [16].

We consider a transition $(q, Z, LU) \Rightarrow_g (q', Z', L'U')$, that is a transition with a guard but no reset. Suppose that we have updated $L'U'$ and now we want our `newbounds` function to compute $L_{new}U_{new}$. In the constant propagation algorithm of [14], we would have set $L_{new}U_{new}$ to be the maximum over LU, $L'U'$, and the constant present in the guard. This is sufficient to maintain Invariant 2. However, it is not necessary to always take the guard g into consideration for the propagation.

Let L_gU_g be the bound function induced by the guard g. In our case, as there is only one constraint, there is only one constant associated to a single clock by L_gU_g. It can be shown that in order to maintain Invariant 2, it suffices to take

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U') & \text{if } [\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z') \text{ or} \\ & \text{if } Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z') \\ \max(LU, L'U', L_gU_g) & \text{otherwise} \end{cases} \tag{2}$$

Since bound propagation is called often in the main algorithm, we need an efficient test for the inclusions in formula (2). The formula requires us to test inclusion w.r.t. $\mathfrak{a}_{\preccurlyeq LU}$ between Z and Z' each time we want to calculate $L_{new}U_{new}$. Although this seems complicated at the first glance, note that Z' is a zone obtained by a successor computation from Z. When we have only a guard in the transition, we have $Z' = \overrightarrow{Z \wedge g}$: in words, zone Z' is obtained by first intersecting Z with g and letting time elapse from the resulting set of valuations. This relation between Z and Z' makes the inclusion test a lot more simpler. We will also see that it is not necessary to consider the inclusion $[\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

Before proceeding, we are obliged to look closer at how zones are represented. Instead of difference bound matrices (DBMs) [11], we will prefer an equivalent representation in terms of distance graphs.

A *distance graph* has clocks as vertices, with an additional special clock x_0 representing the constant 0. For readability, we will often write 0 instead of x_0. Between every two vertices there is an edge with a weight of the form $(\prec, c)$ where $c \in \mathbb{Z}$ and $\prec$ is either $\leq$ or $<$; or $(\prec, c)$ equals $(<, \infty)$. An edge $x \xrightarrow{\prec c} y$ represents a constraint $y - x \prec c$: or in words, the distance from x to y is bounded by c. A distance graph is in *canonical form* if the weight of the edge from x to y is the lower bound of the weights of paths from x to y. A zone Z can be identified with the distance graph in the canonical form representing the constraints in Z. For two clocks x, y we write Z_{xy} for the weight of the edge from x to y in this graph. A special case is when x or y is 0, so for example Z_{0y} denotes the weight of the edge from 0 to y.

We recall a theorem from [15] that yields an efficient test for: $Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

Theorem 2. *Let Z, Z' be two non-empty zones. Then $Z \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$ iff there exist two clocks x, y such that:*

$$Z_{x0} \geq (\leq, -U'_x) \text{ and } Z'_{xy} < Z_{xy} \text{ and } Z'_{xy} + (<, -L'_y) < Z_{x0} \tag{3}$$

We are ready to proceed with our analysis.

Lower bound guard: When the guard on the transition is $w \geqslant d$, the diagonals, that is Z_{xy} with both x, y variables other than zero, do not change during intersection and time-elapse. Hence we have $Z'_{xy} = Z_{xy}$ for such x and y. This shows that (3) cannot be true when both x and y are non-zero as the second condition is false. Yet again, when x is 0, the second condition cannot be true since both $Z_{0y} = Z'_{0y} = (<, \infty)$, as after time-elapse there is no constraint of the form $y \lessdot c$ where $c \in \mathbb{Z}$. It remains to consider the single case when y is 0. It boils down to checking if there exists a clock x such that $Z_{x0} \geq (\leq, -U'_x)$ and $Z'_{x0} < Z_{x0}$. In words, the test asks if there exists a clock x whose label of the edge $x \xrightarrow{\lessdot -c} 0$ in Z has reduced in Z' and additionally the edge weight $(\lessdot, -c)$ in Z satisfies either $c < U'_x$ or $(\lessdot, c) = (\leq, U'_x)$. It can be checked that if $Z \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$, then $[\![g]\!] \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$ too. So in this case Formula (3) simplifies to the following formula with the additional observation that Z'_{x0} can be only lesser than or equal to Z_{x0}.

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } \exists x.\ \big(Z_{x0} \geq (\leq, -U'_x)\big) \wedge \big((Z'_{x0} < Z_{x0})\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases}$$

Note that this test can be easily extended to an incremental procedure: whenever we modify the U' value of a clock, we need to check only this clock. The above definition also suggests that whenever only L' is modified we don't have to check anything and just propagate the new values of L'.

Upper bound guard: When we have an upper bound guard, the diagonals might change. However no edge $0 \to x$ or $x \to 0$ changes. Therefore we need to check (3) for two non-zero variables x and y.

In other words, among clocks x that have a finite U' constant and clocks y that have a finite L' constant, we check if there is a diagonal $x \to y$ that has strictly reduced in Z' and additionally satisfies $Z'_{xy} + (<, L_y) < Z_{x0}$. Yet again, it can be checked that if $Z \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$, then $[\![g]\!] \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. Therefore it is sufficient to check (3) for non zero variables x and y. This gives the following formula function:

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } \exists x, y. \text{ such that} \\ & \big(Z_{x0} \geq (\leq, -U'_x)\big) \ \wedge \ \big(Z'_{xy} < Z_{xy}\big) \wedge \\ & \quad \big(Z'_{xy} + (<, -L'_y) < Z_{x0}\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases} \tag{4}$$

This test can also be done incrementally. Each time we propagate, we need to perform extra checks only when a new clock has got a finite value for either L' or U'.

Other types of transitions with upper bound guard and resets, and multiple guards are treated in the full version of the paper [16]. The pseudocode of the `newbounds` function implementing the obtained tests is presented in the full

version too. Here, we just state the theorem expressing the correctness of the construction.

Theorem 3. *Let v' be a node of ASG and v be its parent. Let $(v.q, v.Z) \Rightarrow_t (v'.q, v'.Z')$ be a transition. Given bound functions $v'.LU$, the functions $L_{new}U_{new}$ as computed by* `newbounds`(v, v', X'_L, X'_U) *satisfy invariant I2, namely:*

$$\mathrm{Post}_t(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq v'.LU}(Z').$$

6 Examples

In this section we will analyze the behavior of our algorithm on some examples in order to explain some of the sources of the gains reported in the experiments of the next section.

The invariants in the definition of adaptive simulation graph (Definition 6) sometimes allow for much smaller LU-bounds than that obtained by static analysis. A very basic example is when during exploration the algorithm does not encounter a node with a disabled edge. In this case all LU-bounds are simply $-\infty$, since propagation does not change such bounds. If LU bounds are $-\infty$, and Z is nonempty then $\mathfrak{a}_{\preccurlyeq LU}(Z)$ is the set of all valuations. So in this case the ASG is just a subgraph of the automaton. We will now see an example where such a situation occurs and yields exponential gain over the static analysis method used by UPPAAL, and the on-the-fly constant propagation algorithm from [14].

Consider the automaton $\mathcal{D}_n$ shown in Figure 1. This is a slightly modified example from [17]. We have changed all guards to check for an equality. Automaton $\mathcal{D}_n$ is a parallel composition of three components. The first two components respectively reset the x-clocks and y-clocks. The third component can be fired only after the first two have reached their a_n states. The reachable states of the product automaton $\mathcal{D}_n$ are of the form (a_i, a_j, b_0) and (a_n, a_n, b_k) where $i, j, k \in \{0, \ldots, n\}$. Let us assume that no state is accepting so that any forward exploration algorithm should explore the entire search space.

Clearly, all the transitions can be fired if no time elapses in the states (a_i, a_j, b_0) for $i, j \in 1, \ldots, n-1$, and exactly one time unit elapses in (a_n, a_n, b_0). Therefore, an ASG for $\mathcal{D}_n$ will have no edges disabled which implies that in each node the LU-constants given by our algorithm are $-\infty$. The number of uncovered nodes in the ASG constructed by our algorithm will be the same as the number of states.

However, the static analysis procedure would give $L = U = 1$ for every clock. It can be proved that this would yield a zone graph with at least 2^n nodes. As all the edges are enabled, the constant propagation algorithm from [14] would explore a path up to (a_n, a_n, b_n). This would therefore give $L = U = 1$ for each clock, similar to static analysis. So in this case too there would be at least 2^n uncovered nodes in the reachability tree obtained.

Let us now see an example when there is a disabled edge. Consider the automaton $\mathcal{A}_2$ in Figure 1. One can see that the last transition with the upper bound is not fireable, and that the reason is the guard $x \geq 5$. Our algorithm

would say that at q_0 the relevant constants are: $L_0(x) = 5$ and $U_0(x) = 1$ and the rest are $-\infty$. The static analysis algorithm or the constant propagation would give additionally $L(y) = 5$ and $L(z) = 100$, which is unnecessary. We will now see that this pruning can sometimes lead to an exponential gain.

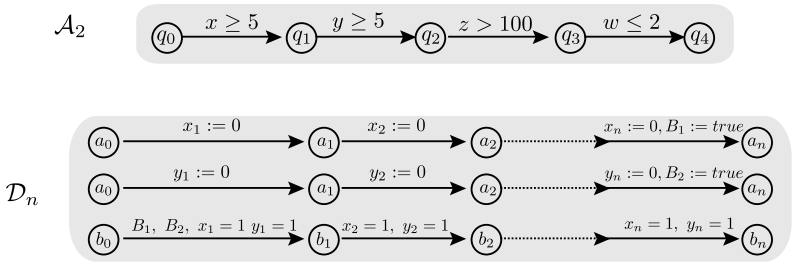

Fig. 1. Automata $\mathcal{A}_2$ and $\mathcal{D}_n$

Consider automaton $\mathcal{D}'_n$ obtained from $\mathcal{D}_n$ in Figure 1 by changing every constraint involving the y-clock to $y = 2$. If the algorithm is "fortunate" to choose the right order of resets, it can reach node (a_n, a_n, b_n) without seeing a disabled edge. Due to this it will construct an ASG with the number of uncovered nodes equal to number of states of the automaton.

If it is not the case, then it reaches the state (a_n, a_n, b_0) with a zone where there is an i such that $y_i \leq x_i$ and for all $j < i$, $x_i \leq y_i$. From such a zone, the path can be taken till b_{i-1} after which the transition gets disabled because we check for $y_i \geq 2$ and $x_i \leq 1$. The disabled edge gives the constant $U(x_i) = 1$ and the propagation algorithm additionally generates $L(y_i) = 2$. But it generates no bounds for other clocks. In the result, the reached node will cover any other node $((a_n, a_n, b_0), Z', L'U')$ with Z' satisfying $x_i \leq y_i$. So there will be at most n uncovered nodes with the state (a_n, a_n, b_0) and hence the total number of uncovered nodes will be at most quadratic in n. In fact, it will be linear but for this a more careful analysis is needed.

The static analysis procedure would give $L = U = 2$ for all y-clocks and $L = U = 1$ for all x-clocks. It can be shown that this would result in at least 2^n uncovered nodes with state (a_n, a_n, b_0).

The on-the-fly propagation algorithm [14] could work slightly differently from the previous case. The constants generated depend on the first path. If the first path leads up to (a_n, a_n, b_n) then there are constants generated for all clocks. Then, the zone cannot cover any of the future zones that appear at (a_n, a_n, b_n). A depth-first search algorithm would clearly then be exponential. Otherwise, if the path gets cut at b_{k-1} constants are generated for all clocks $x_1, y_1, \ldots, x_k, y_k$. In this case, at least 2^k nodes at (a_n, a_n, b_0) need to be distinguished.

7 Experiments

We report experiments in Table 1 for classical benchmarks from the literature. The first two columns compare UPPAAL 4.1.13 with our own implementation

Table 1. Comparison of reachability algorithms: number of visited nodes and running time. For each model and each algorithm, we kept the best of depth-first search and breadth-first search. Experiments done on a MacBook with 2.4GHz Intel Core Duo processor and 2GB of memory running MacOS X 10.6.8. Missing numbers are due to time out (150s) or memory out (1Gb).

Model	nb. of	UPPAAL (-C)		$Extra^+_{LU}$,sa		$\mathfrak{a}_{\preccurlyeq LU}$,otf		$\mathfrak{a}_{\preccurlyeq LU}$,disabled	
	clocks	nodes	sec.	nodes	sec.	nodes	sec.	nodes	sec.
$\mathcal{D}''_7$	14	18654	11.6	18654	8.1	213	0.0	72	0.0
$\mathcal{D}''_8$	16					274	0.0	90	0.0
$\mathcal{D}''_{70}$	140							5112	1.9
CSMA/CD 10	11	120845	1.9	120844	6.3	78604	6.1	51210	4.0
CSMA/CD 11	12	311310	5.4	311309	16.8	198669	16.1	123915	10.2
CSMA/CD 12	13	786447	14.8	786446	44.0	493582	41.8	294924	25.2
FDDI 50	151	12605	52.9	12606	29.4	5448	14.7	401	0.8
FDDI 70	211							561	2.7
FDDI 140	421							1121	40.6
Fischer 9	9	135485	2.4	135485	8.9	135485	11.4	135485	14.8
Fischer 10	10	447598	10.1	447598	34.0	447598	42.8	447598	56.8
Fischer 11	11	1464971	40.4	1464971	126.8				
Stari 2	7	7870	0.1	6993	0.4	5779	0.4	4305	0.4
Stari 3	10	136632	1.7	113958	9.4	82182	8.2	43269	4.5
Stari 4	13	1323193	26.2	983593	109.0	602762	84.9	296982	41.5

of UPPAAL's algorithm ($Extra^+_{LU}$,sa). We have taken particular care to ensure that the two implementations deal with the same model and explore it in the same way. However, on the last example (Stari), we did not manage to force the same search order in the two tools.

The last two algorithms are using bounds propagation. In the third column ($\mathfrak{a}_{\preccurlyeq LU}$,otf), we report results for the algorithm in [14] that propagates the bounds from every transition (enabled or disabled) that is encountered during the exploration of the zone graph. Since this algorithm only considers the bounds that are reachable in the zone graph, it generally visits less nodes than UPPAAL's algorithm. The last column ($\mathfrak{a}_{\preccurlyeq LU}$,disabled) corresponds to the algorithm introduced in this paper. It propagates the bounds that come from the disabled transitions only. As a result it generally outperforms the other algorithms. The actual implementation of our algorithm is slightly more sophisticated than the one presented in Section 4. Like UPPAAL, it uses a Passed/Waiting list instead of a stack. The implemented algorithm is presented in the Appendix of [16].

The results show a huge gain on two examples: $\mathcal{D}''$ and FDDI. D''_n corresponds to the automaton $\mathcal{D}_n$ in Fig. 1 where the tests $x_k = 1, y_k = 1$ have been replaced by $(0 < x_k \leq 1), (1 < y_k \leq 2)$. While it was easier in Section 6 to analyze the example with equality tests, we wanted here to show that the same performance gain occurs also when static L bounds are different from static U bounds. The number of nodes visited by algorithm $\mathfrak{a}_{\preccurlyeq LU}$,disabled exactly corresponds to the number of states in the timed automaton. The situation with the FDDI example is similar: it has only one disabled transition. The other three algorithms take useless clock bounds into account. As a result they quickly face a combinatorial explosion in the number of visited nodes. We managed to analyze $\mathcal{D}''_n$ up to $n = 70$ and FDDI up to size 140 despite the huge number of clocks.

Fischer example represents the worst case scenario for our algorithm. Dynamic bounds calculated by algorithms $\mathfrak{a}_{\preccurlyeq_{LU}}$,otf and $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled turn out to be the same LU-bounds given by static analysis.

The remaining two models, CSMA/CD and Stari [8] show the average situation. The interest of Stari is that it is a very complex example with both a big discrete part and a big continuous part. The model is exactly the one presented in op. cit. but for a fixed initial state. Algorithm $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled discards many clock bounds by considering disabled transitions only. This leads to a significant gain in the number of visited nodes at a reasonable cost.

8 Conclusions

We have pursued an idea of adapting abstractions while searching through the reachability space of a timed automaton. Our objective has been to obtain as low LU-bounds as possible without sacrificing practicability of the approach. In the end, the experimental results show that algorithm $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled improves substantially the state-of-the art forward exploration algorithms for the reachability problem in timed automata.

At first sight, a more refined approach would be to work with constraints themselves instead of LU-abstractions. Following the pattern presented here, when encountering a disabled transition, one could take a constraint that makes it disabled, and then propagate this constraint backwards using, say, weakest precondition operation. A major obstacle in implementing this approach is the covering condition, like G3 in our case. When a node is covered, a loop is formed in the abstract system. To ensure soundness, the abstraction in a covered node should be an invariant of this loop. A way out of this problem can be to consider a different covering condition as proposed by McMillan [18], but then this condition requires to develop the abstract model much more than we do. So from this perspective we can see that LU-bounds are a very interesting tool to get a loop invariant cheaply, and offer a good balance between expressivity and algorithmic effectiveness.

We do not make any claim about optimality of our backward propagation algorithm. For example, one can see that it gives different results depending on the order of treating the constraints. Even for a single constraint, our algorithm is not optimal in a sense that there are examples when we could obtain smaller LU-bounds. At present we do not know if it is possible to compute optimal LU-bounds efficiently. In our opinion though, it will be even more interesting to look at ways of cleverly rearranging transitions of an automaton to limit bounds propagation even further. Another promising improvement is to introduce some partial order techniques, like parallelized interleaving from [19]. We think that the propagation mechanisms presented here are well adapted to such methods.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126(2), 183–235 (1994)
2. Alur, R., Itai, A., Kurshan, R.P., Yannakakis, M.: Timing verification by successive approximation. Inf. Comput. 118(1), 142–157 (1995)

3. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)
4. Behrmann, G., Bouyer, P., Larsen, K.G., Pelanek, R.: Lower and upper bounds in zone-based abstractions of timed automata. Int. Journal on Software Tools for Technology Transfer 8(3), 204–215 (2006)
5. Behrmann, G., David, A., Larsen, K.G., Haakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society (2006)
6. Bengtsson, J.E., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
7. Bouyer, P., Laroussinie, F., Reynier, P.-A.: Diagonal constraints in timed automata: Forward analysis of timed systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 112–126. Springer, Heidelberg (2005)
8. Bozga, M., Maler, O., Tripakis, S.: Efficient verification of timed automata using dense and discrete time semantics. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 125–141. Springer, Heidelberg (1999)
9. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. Form. Methods Syst. Des. 1(4), 385–415 (1992)
10. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
11. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
12. Dill, D.L., Wong-Toi, H.: Verification of real-time systems by successive over and under approximation. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 409–422. Springer, Heidelberg (1995)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
14. Herbreteau, F., Kini, D., Srivathsan, B., Walukiewicz, I.: Using non-convex approximations for efficient analysis of timed automata. In: FSTTCS. LIPIcs, vol. 13, pp. 78–89 (2011)
15. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. In: LICS, pp. 375–384 (2012)
16. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. arXiv:1301.3127, Extended version with proofs (2013)
17. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. TCS 345(1), 27–59 (2005)
18. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
19. Morbé, G., Pigorsch, F., Scholl, C.: Fully symbolic model checking for timed automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 616–632. Springer, Heidelberg (2011)
20. Sorea, M.: Lazy approximation for dense real-time systems. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004)
21. Wang, F.: Efficient verification of timed automata with BDD-like data structures. Int. J. Softw. Tools Technol. Transf. 6(1), 77–97 (2004)
22. Wong-Toi, H.: Symbolic Approximations of Verifying Real-Time Systems. PhD thesis, Stanford University (March 1995)