# Using non-convex approximations for efficient analysis of timed automata

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

1 **Université de Bordeaux, IPB, CNRS, LaBRI UMR5800**
2 **Indian Institute of Technology Bombay, Department of Computer Science and Engineering**

## Abstract

The reachability problem for timed automata asks if there exists a path from an initial state to a target state. The standard solution to this problem involves computing the zone graph of the automaton, which in principle could be infinite. In order to make the graph finite, zones are approximated using an extrapolation operator. For reasons of efficiency in current algorithms extrapolation of a zone is always a zone; and in particular it is convex.

In this paper, we propose to solve the reachability problem without such extrapolation operators. To ensure termination, we provide an efficient algorithm to check if a zone is included in the so called region closure of another. Although theoretically better, closure cannot be used in the standard algorithm since a closure of a zone may not be convex.

An additional benefit of the proposed approach is that it permits to calculate approximating parameters on-the-fly during exploration of the zone graph, as opposed to the current methods which do it by a static analysis of the automaton prior to the exploration. This allows for further improvements in the algorithm. Promising experimental results are presented.

**1998 ACM Subject Classification** D.2.4 Software/Program Verification; F.3.1 Specifying and Verifying and Reasoning about Programs; F.4.1 Mathematical Logic

**Keywords and phrases** Timed Automata; Model-checking; Non-convex abstraction; On-the-fly abstraction

**Digital Object Identifier** 10.4230/LIPIcs.FSTTCS.2011.78

## 1 Introduction

Timed automata [1] are obtained from finite automata by adding clocks that can be reset and whose values can be compared with constants. The crucial property of timed automata is that their reachability problem is decidable: one can check if a given target state is reachable from the initial state. Reachability algorithms are at the core of verification tools like Uppaal [4] or RED [16], and are used in industrial case studies [11, 6]. The standard solution constructs a search tree whose nodes are approximations of zones. In this paper we give an efficient algorithm for checking if a zone is included in an approximation of another zone. This enables a reachability algorithm to work with search trees whose nodes are just unapproximated zones. This has numerous advantages: one can use non-convex approximations, and one can compute approximating parameters on the fly.

The first solution to the reachability problem has used *regions*, which are equivalence classes of clock valuations. Subsequent research has shown that the region abstraction is very inefficient and an other method using *zones* instead of regions has been proposed. This can be implemented efficiently using DBMs [10] and is used at present in almost all timed-verification tools. The number of reachable zones can be infinite, so one needs an abstraction operator to get a finite approximation. The simplest is to approximate a zone with the set of regions it intersects, the so called *closure* of a zone. Unfortunately, the closure may not always be convex and no efficient representation of closures is known. For this reason implementations use another convex approximation that is also based on (refined) regions.

We propose a new algorithm for the reachability problem using closures of zones. To this effect we provide an efficient algorithm for checking whether a zone is included in a closure of another zone. In consequence we can work with non-convex approximations without a need to store them explicitly.

Thresholds for approximations are very important for efficient implementation. Good thresholds give substantial gains in time and space. The simplest approach is to take as a threshold the maximal constant appearing in a transition of the automaton. A considerable gain in efficiency can be obtained by analyzing the graph of the automaton and calculating thresholds specific for each clock and state of the automaton [2]. An even more efficient approach is the so called LU-approximation that distinguishes between upper and lower bounds [3]. This is the method used in the current implementation of UPPAAL. We show that we can accommodate closure on top of the LU-approximation at no extra cost.

Since our algorithm never stores approximations, we can compute thresholds on-the-fly. This means that our computation of thresholds does not take into account unreachable states. In consequence in some cases we get much better LU-thresholds than those obtained by static analysis. This happens in particular in a very common context of analysis of parallel compositions of timed automata.

## Related work

The topic of this paper is approximation of zones and efficient handling of them. We show that it is possible to use non-convex approximations and that it can be done efficiently. In particular, we improve on state of the art approximations [3]. Every forward algorithm needs approximations, so our work can apply to tools like RED or UPPAAL.

Recent work [15] reports on backward analysis approach using general linear constraints. This approach does not use approximations and relies on SMT solver to simplify the constraints. Comparing forward and backward methods would require a substantial test suite, and is not the subject of this paper.

## Organization of the paper

The next section presents the basic notions and recalls some of their properties. Section 3 describes the new algorithm for efficient inclusion test between a zone and a closure of another zone. The algorithm constructing the search tree and calculating approximations on-the-fly is presented in Section 4. Some results obtained with a prototype implementation are presented in the last section. All missing proofs are presented in the full version of the paper [13].

## 2 Preliminaries

### 2.1 Timed automata 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 *clock constraint* 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 $\nu : X \to \mathbb{R}_{\geq 0}$. We denote $\mathbb{R}_{\geq 0}^X$ the set of clock valuations over $X$, and $\mathbf{0}$ the valuation that associates 0 to every clock in $X$. We write $\nu \vDash \phi$ when $\nu$ satisfies $\phi \in \Phi(X)$, i.e. when every constraint in $\phi$ holds after replacing every $x$ by $\nu(x)$. For $\delta \in \mathbb{R}_{\geq 0}$, let $\nu + \delta$ be the valuation that associates $\nu(x) + \delta$ to every clock $x$. For $R \subseteq X$, let $[R]\nu$ be the valuation that sets $x$ to 0 if $x \in R$, and that sets $x$ to $\nu(x)$ 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. An example of a TA is depicted in Figure 1. The class of TA we consider is commonly known as diagonal-free TA since clock comparisons like $x - y \leq 1$ are disallowed. Notice that since we are interested in state reachability, considering timed automata without state invariants does not entail any loss of generality. Indeed, state invariants can be added to guards, then removed, while preserving state reachability.

A *configuration* of $\mathcal{A}$ is a pair $(q, \nu) \in Q \times \mathbb{R}_{\geq 0}^X$; $(q_0, \mathbf{0})$ is the *initial configuration*. We write $(q, \nu) \xrightarrow{\delta,t} (q', \nu')$ if there exists $\delta \in \mathbb{R}_{\geq 0}$ and a transition $t = (q, g, R, q')$ in $\mathcal{A}$ such that $\nu + \delta \vDash g$, and $\nu' = [R]\nu$. Then $(q', \nu')$ is called a *successor* of $(q, \nu)$. A *run* of $\mathcal{A}$ is a finite sequence of transitions: $(q_0, \nu_0) \xrightarrow{\delta_0,t_0} (q_1, \nu_1) \xrightarrow{\delta_1,t_1} \cdots (q_n, \nu_n)$ starting from $(q_0, \nu_0) = (q_0, \mathbf{0})$.

A run is *accepting* if it ends in a configuration $(q_n, \nu_n)$ with $q_n \in Acc$. The *reachability problem* is to decide whether a given automaton has an accepting run. This problem is known to be Pspace-complete [1, 8].

## 2.2 Symbolic semantics for timed automata

The reachability problem is solved using so-called symbolic semantics. It considers sets of (uncountably many) valuations instead of valuations separately. A *zone* is a set of valuations defined by a conjunction of two kinds of constraints: comparison of difference between two clocks with an integer like $x - y \# c$, or comparison of a single clock with an integer like $x \# c$, where $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$. For instance $(x - y \geq 1) \wedge (y < 2)$ is a zone. The transition relation on valuations is transferred to zones as follows. We have $(q, Z) \xrightarrow{t} (q', Z')$ if $Z'$ is the set of valuations $\nu'$ such that $(q, \nu) \xrightarrow{\delta,t} (q', \nu')$ for some $\nu \in Z$ and $\delta \in \mathbb{R}_{\geq 0}$. The node $(q', Z')$ is called a successor of $(q, Z)$. It can be checked that if $Z$ is a zone, then $Z'$ is also a zone.

The *zone graph* of $\mathcal{A}$, denoted $ZG(\mathcal{A})$, has nodes of the form $(q, Z)$ with initial node $(q_0, \{\mathbf{0}\})$, and edges defined as above. Immediately from the definition of $ZG(\mathcal{A})$ we infer that $\mathcal{A}$ has an accepting run iff there is a node $(q, Z)$ reachable in $ZG(\mathcal{A})$ with $q \in Acc$.

Now, every node $(q, Z)$ has finitely many successors: at most one successor of $(q, Z)$ per transition in $\mathcal{A}$. Still a reachability algorithm may not terminate as the number of reachable nodes in $ZG(\mathcal{A})$ may not be finite [9]. The next step is thus to define an abstract semantics of $\mathcal{A}$ as a finite graph. The basic idea is to define a finite partition of the set of valuations $\mathbb{R}_{\geq 0}^X$. Then, instead of considering nodes $(q, S)$ with set of valuations $S$ (e.g. zones $Z$), one considers a union of the parts of $\mathbb{R}_{\geq 0}^X$ that intersect $S$. This gives the finite abstraction.

Let us consider a *bound function* associating to each clock $x$ of $\mathcal{A}$ a bound $\alpha_x \in \mathbb{N}$. A *region* [1] with respect to $\alpha$ is the set of valuations specified as follows:

1. for each clock $x \in X$, one constraint from the set:

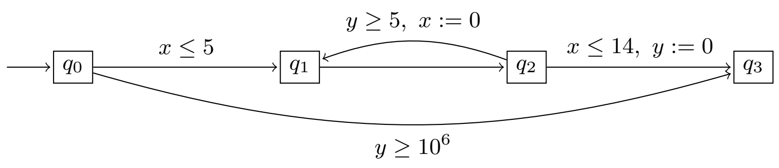

**Figure 1** Timed automaton $\mathcal{A}$.

$$\{x = c \mid c = 0, \dots, \alpha_x\} \cup \{c - 1 < x < c \mid c = 1, \dots, \alpha_x\} \cup \{x > \alpha_x\}$$

2. for each pair of clocks $x, y$ having interval constraints: $c - 1 < x < c$ and $d - 1 < y < d$, it is specified if $fract(x)$ is less than, equal to or greater than $fract(y)$.

It can be checked that the set of regions is a finite partition of $\mathbb{R}^X_{\geq 0}$.

The *closure abstraction* of a set of valuations $S$, denoted $Closure_\alpha(S)$, is the union of the regions that intersect $S$ [7]. A simulation graph, denoted $SG_\alpha(\mathcal{A})$, has nodes of the form $(q, S)$ where $q$ is a state of $\mathcal{A}$ and $S \subseteq \mathbb{R}^X_{\geq 0}$ is a set of valuations. The initial node of $SG_\alpha(\mathcal{A})$ is $(q_0, \{\mathbf{0}\})$. There is an edge $(q, S) \xrightarrow{t} (q', Closure_\alpha(S'))$ in $SG_\alpha(\mathcal{A})$ iff $S'$ is the set of valuations $\nu'$ such that $(q, \nu) \xrightarrow{\delta,t} (q', \nu')$ for some $\nu \in S$ and $\delta \in \mathbb{R}_{\geq 0}$. Notice that the reachable part of $SG_\alpha(\mathcal{A})$ is finite since the number of regions is finite.

The definition of the graph $SG_\alpha(\mathcal{A})$ is parametrized by a bound function $\alpha$. It is well-known that if we take $\alpha_\mathcal{A}$ associating to each clock $x$ the maximal integer $c$ such that $x \# c$ appears in some guard of $\mathcal{A}$ then $SG_\alpha(\mathcal{A})$ preserves the reachability properties.

▶ **Theorem 1.** *[7] $\mathcal{A}$ has an accepting run iff there is a reachable node $(q, S)$ in $SG_\alpha(\mathcal{A})$ with $q \in Acc$ and $\alpha_\mathcal{A} \leq \alpha$.*

For efficiency it is important to have a good bound function $\alpha$. The nodes of $SG_\alpha(\mathcal{A})$ are unions of regions. Hence the size of $SG_\alpha(\mathcal{A})$ depends on the number of regions which is $\mathcal{O}\big(|X|!.2^{|X|}.\prod_{x \in X}(2.\alpha_x + 2)\big)$ [1]. It follows that smaller values for $\alpha$ yield a coarser, hence smaller, symbolic graph $SG_\alpha(\mathcal{A})$. Note that current implementations do not use closure but some convex under-approximation of it that makes the graph even bigger.

It has been observed in [2] that instead of considering a global bound function $\alpha_\mathcal{A}$ for all states in $\mathcal{A}$, one can use different functions in each state of the automaton. Consider for instance the automaton $\mathcal{A}$ in Figure 1. Looking at the guards, we get that $\alpha_x = 14$ and $\alpha_y = 10^6$. Yet, a closer look at the automaton reveals that in the state $q_2$ it is enough to take the bound $\alpha_y(q_2) = 5$. This observation from [2] points out that one can often get very big gains by associating a bound function $\alpha(q)$ to each state $q$ in $\mathcal{A}$ that is later used for the abstraction of nodes of the form $(q, Closure_{\alpha(q)}(S))$. In op. cit. an algorithm for inferring bounds based on static analysis of the structure of the automaton is proposed. In Section 4.2 we will show how to calculate these bounds on-the-fly during the exploration of the automaton's state space.

## 3 Efficient testing of inclusion in a closure of a zone

The tests of the form $Z \subseteq Closure_\alpha(Z')$ will be at the core of the new algorithm we propose. This is an important difference with respect to the standard algorithm that makes the tests of the form $Z \subseteq Z'$. The latter tests are done in $\mathcal{O}(|X|^2)$ time, where $|X|$ is the number of clocks. We present in this section a simple algorithm that can do the tests $Z \subseteq Closure_\alpha(Z')$ at the same complexity with neither the need to represent nor to compute the closure.

We start by examining the question as to how one decides if a region $R$ intersects a zone $Z$. The important point is that it is enough to verify that the projection on every pair of variables is nonempty. This is the cornerstone for the efficient inclusion testing algorithm that even extends to LU-approximations.

## 3.1 When is $R \cap Z$ empty

It will be very convenient to represent zones by *distance graphs*. Such a graph has clocks as vertices, with an additional special clock $x_0$ representing 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 $(\preccurlyeq, c)$ where $c \in \mathbb{Z} \cup \{\infty\}$ and $\preccurlyeq$ is either $\leq$ or $<$. An edge $x \xrightarrow{\preccurlyeq c} y$ represents a constraint $y - x \preccurlyeq c$: or in words, the distance from $x$ to $y$ is bounded by $c$. Let $[\![G]\!]$ be the set of valuations of clock variables satisfying all the constraints given by the edges of $G$ with the restriction that the value of $x_0$ is 0.

An arithmetic over the weights $(\preccurlyeq, c)$ can be defined as follows [5].

*Equality* $(\preccurlyeq_1, c_1) = (\preccurlyeq_2, c_2)$ if $c_1 = c_2$ and $\preccurlyeq_1 = \preccurlyeq_2$.
*Addition* $(\preccurlyeq_1, c_1) + (\preccurlyeq_2, c_2) = (\preccurlyeq, c_1 + c_2)$ where $\preccurlyeq = <$ iff either $\preccurlyeq_1$ or $\preccurlyeq_2$ is $<$.
*Minus* $-(\preccurlyeq, c) = (\preccurlyeq, -c)$.
*Order* $(\preccurlyeq_1, c_1) < (\preccurlyeq_2, c_2)$ if either $c_1 < c_2$ or ($c_1 = c_2$ and $\preccurlyeq_1 = <$ and $\preccurlyeq_2 = \leq$).
*Floor* $\lfloor(<, c)\rfloor = (\leq, c-1)$ and $\lfloor(\leq, c)\rfloor = (\leq, c)$.

This arithmetic lets us talk about the weight of a path as a weight of the sum of its edges. A cycle in a distance graph $G$ is said to be *negative* if the sum of the weights of its edges is at most $(<, 0)$; otherwise the cycle is *positive*. The following useful proposition is folklore.

▶ **Proposition 2.** *A distance graph $G$ has only positive cycles iff $[\![G]\!] \neq \emptyset$.*

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 *distance graph of a region $R$*, denoted $G_R$, is the canonical graph representing all the constraints defining $R$. Similarly $G_Z$ for a zone $Z$.

We can now state a necessary and sufficient condition for the intersection $R \cap Z$ to be empty in terms of cycles in distance graphs. We denote by $R_{xy}$ the weight of the edge $x \xrightarrow{\preccurlyeq_{xy} c_{xy}} y$ in the canonical distance graph representing $R$. Similarly for $Z$.

▶ **Proposition 3.** *Let $R$ be a region and let $Z$ be a zone. The intersection $R \cap Z$ is empty iff there exist variables $x, y$ such that $Z_{yx} + R_{xy} \leq (<, 0)$.*

A variant of this fact has been proven as an intermediate step of Proposition 2 in [7].

## 3.2 Efficient inclusion testing

Our goal is to efficiently perform the test $Z \subseteq \mathit{Closure}(Z')$ for two zones $Z$ and $Z'$. We are aiming at $\mathcal{O}(|X|^2)$ complexity, since this is the complexity of current algorithms used for checking inclusion of two zones. Proposition 3 can be used to efficiently test the inclusion $R \subseteq \mathit{Closure}(Z')$. It remains to understand what are the regions intersecting the zone $Z$ and then to consider all possible cases. The next lemma basically says that every consistent instantiation of an edge in $G_Z$ leads to a region intersecting $Z$.

▶ **Lemma 4.** *Let $G$ be a distance graph in canonical form, with all cycles positive. Let $x, y$ be two variables, and let $x \xrightarrow{\preccurlyeq_{xy} c_{xy}} y$ and $y \xrightarrow{\preccurlyeq_{yx} c_{yx}} x$ be edges in $G$. For every $d \in \mathbb{R}$ such that $d \preccurlyeq_{xy} c_{xy}$ and $-d \preccurlyeq_{yx} c_{yx}$ there exists a valuation $v \in [\![G]\!]$ with $v(y) - v(x) = d$.*

Thanks to this lemma it is enough to look at edges of $G_Z$ one by one to see what regions we can get. This insight is used to get the desired efficient inclusion test

▶ **Theorem 5.** *Let $Z, Z'$ be zones. Then, $Z \nsubseteq Closure_\alpha(Z')$ iff there exist variables $x$, $y$, both different from $x_0$, such that one of the following conditions hold:*

1. *$Z'_{0x} < Z_{0x}$ and $Z'_{0x} \leq (\leq, \alpha_x)$, or*
2. *$Z'_{x0} < Z_{x0}$ and $Z_{x0} \geq (\leq, -\alpha_x)$, or*
3. *$Z_{x0} \geq (\leq, -\alpha_x)$ and $Z'_{xy} < Z_{xy}$ and $Z'_{xy} \leq (\leq, \alpha_y) + \lfloor Z_{x0} \rfloor$.*

## Comparison with the algorithm for $Z \subseteq Z'$

Given two zones $Z$ and $Z'$, the procedure for checking $Z \subseteq Z'$ works on two graphs $G_Z$ and $G_{Z'}$ that are in canonical form. This form reduces the inclusion test to comparing the edges of the graphs one by one. Note that our algorithm for $Z \subseteq Closure_\alpha(Z')$ does not do worse. It works on $G_Z$ and $G_{Z'}$ too. The edge by edge checks are only marginally more complicated. The overall procedure is still $\mathcal{O}(|X|^2)$.

## 3.3 Handling LU-approximation

In [3] the authors propose to distinguish between maximal constants used in upper and lower bounds comparisons: for each clock $x$, $L_x \in \mathbb{N} \cup \{-\infty\}$ represents the maximal constant $c$ such that there exists a constraint $x > c$ or $x \geq c$ in a guard of a transition in the automaton; dually, $U_x \in \mathbb{N} \cup \{-\infty\}$ represents the maximal constant $c$ such that there is a constraint $x < c$ or $x \leq c$ in a guard of a transition. If such a $c$ does not exist, then it is considered to be $-\infty$. They have introduced an extrapolation operator $Extra^+_{LU}(Z)$ that takes into account this information. This is probably the best presently known convex abstraction of zones.

We now explain how to extend our inclusion test to handle LU approximation, namely given $Z$ and $Z'$ how to directly check $Z \subseteq Closure_\alpha(Extra^+_{LU}(Z'))$ efficiently. Observe that for each $x$, the maximal constant $\alpha_x$ is the maximum of $L_x$ and $U_x$. In the sequel, this is denoted $Z \subseteq Closure^+_{LU}(Z')$. For this we need to understand first when a region intersecting $Z$ intersects $Extra^+_{LU}(Z')$. Therefore, we study the conditions that a region $R$ should satisfy if it intersects $Extra^+_{LU}(Z)$ for a zone $Z$.

We recall the definition given in [3] that has originally been presented using difference bound matrices (DBM). In a DBM $(c_{ij}, \prec_{i,j})$ stands for $x_i - x_j \prec_{i,j} c_{i,j}$. In the language of distance graphs, this corresponds to an edge $x_j \xrightarrow{\prec_{i,j} c_{i,j}} x_i$; hence to $Z_{ji}$ in our notation. Let $Z^+$ denote $Extra^+_{LU}(Z)$ and $G_{Z^+}$ its distance graph. We have:

$$Z^+_{xy} = \begin{cases} (<, \infty) & \text{if } Z_{xy} > (\leq, L_y) \\ (<, \infty) & \text{if } -Z_{y0} > (\leq, L_y) \\ (<, \infty) & \text{if } -Z_{x0} > (\leq, U_x), y \neq 0 \\ (<, -U_x) & \text{if } -Z_{x0} > (\leq, U_x), y = 0 \\ Z_{xy} & \text{otherwise.} \end{cases} \tag{1}$$

From this definition it will be important for us to note that $G_{Z^+}$ is $G_Z$ with some weights put to $(<, \infty)$ and some weights on the edges to $x_0$ put to $(<, -U_x)$. Note that $Extra^+_{LU}(Z')$ is not in the canonical form. If we put $Extra^+_{LU}(Z')$ into the canonical form then we could just use Theorem 5. We cannot afford to do this since canonization can take cubic time [5]. The following theorem implies that we can do the test without canonizing $Extra^+_{LU}(Z')$. Hence we can get a simple quadratic test also in this case.

▶ **Theorem 6.** *Let $Z, Z'$ be zones. Let $Z'^{+}$ denote $Extra^{+}_{LU}(Z')$ obtained from $Z'$ using Equation 1 for each edge. Note that $Z'^{+}$ is not necessarily in canonical form. Then, we get that $Z \not\subseteq Closure_{\alpha}(Z'^{+})$ iff there exist variables $x$, $y$ different form $x_0$ such that one of the following conditions hold:*

1. $Z'^{+}_{0x} < Z_{0x}$ *and* $Z'^{+}_{0x} \leq (\leq, \alpha_x)$*, or*
2. $Z'^{+}_{x0} < Z_{x0}$ *and* $Z_{x0} \geq (\leq, -\alpha_x)$*, or*
3. $Z_{x0} \geq (\leq, -\alpha_x)$ *and* $Z'^{+}_{xy} < Z_{xy}$ *and* $Z'^{+}_{xy} \leq (\leq, \alpha_y) + \lfloor Z_{x0} \rfloor$.

## 4 A New Algorithm for Reachability

Our goal is to decide if a final state of a given timed automaton is reachable. We do it by computing a finite prefix of the reachability tree of the zone graph $ZG(\mathcal{A})$ that is sufficient to solve the reachability problem. Finiteness is ensured by not exploring a node $(q, Z)$ if there exists a $(q, Z')$ such that $Z \subseteq Closure_{\alpha}(Z')$, for a suitable $\alpha$. We will first describe a simple algorithm based on the closure and then we will address the issue of finding tighter bounds for the clock values.

### 4.1 The basic algorithm

Given a timed automaton $\mathcal{A}$ we first calculate the bound function $\alpha_{\mathcal{A}}$ as described just before Theorem 1. Each node in the tree that we compute is of the form $(q, Z)$, where $q$ is a state of the automaton, and $Z$ is an unapproximated zone. The root node is $(q_0, Z_0)$, which is the initial node of $ZG(\mathcal{A})$. The algorithm performs a depth first search: at a node $(q, Z)$, a transition $t = (q, g, r, q')$ not yet considered for exploration is picked and the successor $(q', Z')$ is computed where $(q, Z) \xrightarrow{t} (q', Z')$ in $ZG(\mathcal{A})$. If $q'$ is a final state and $Z'$ is not empty then the algorithm terminates. Otherwise the search continues from $(q', Z')$ unless there is already a node $(q', Z'')$ with $Z' \subseteq Closure_{\alpha_{\mathcal{A}}}(Z'')$ in the current tree.

The correctness of the algorithm is straightforward. It follows from the fact that if $Z' \subseteq Closure_{\alpha_{\mathcal{A}}}(Z'')$ then all the states reachable from $(q', Z')$ are reachable from $(q', Z'')$ and hence it is not necessary to explore the tree from $(q', Z')$. Termination of the algorithm is ensured since there are finitely many sets of the form $Closure_{\alpha_{\mathcal{A}}}(Z)$. Indeed, the algorithm will construct a prefix of the reachability tree of $SG_{\alpha}(\mathcal{A})$ as described in Theorem 1.

The above algorithm does not use the classical extrapolation operator named $Extra^{+}_{M}$ in [3] and $Extra^{+}_{\alpha}$ hereafter, but the coarser $Closure_{\alpha}$ operator [7]. This is possible since the algorithm does not need to represent $Closure_{\alpha}(Z)$, which is in general not a zone. Instead of storing $Closure_{\alpha}(Z)$ the algorithm just stores $Z$ and performs tests $Z \subseteq Closure_{\alpha}(Z')$ each time it is needed (in contrast to Algorithm 2 in [7]). This is as efficient as testing $Z \subseteq Z'$ thanks to the algorithm presented in the previous section.

Since $Closure_{\alpha}$ is a coarser abstraction, this simple algorithm already covers some of the optimizations of the standard algorithm. For example the $Extra^{+}_{\alpha}(Z)$ abstraction proposed in [3] is subsumed since $Extra^{+}_{\alpha}(Z) \subseteq Closure_{\alpha}(Z)$ for any zone $Z$ [7, 3]. Other important optimizations of the standard algorithm concern finer computation of bounding functions $\alpha$. We now show that the structure of the proposed algorithm allows to improve this too.

### 4.2 Computing clock bounds on-the-fly

We can improve on the idea of Behrmann et al. [2] of computing a bound function $\alpha_q$ for each state $q$. We will compute these bounding functions on-the-fly and they will depend also on a zone and not just a state. An obvious gain is that we will never consider constraints

**Algorithm 1** Reachability algorithm with on-the-fly bound computation and non-convex abstraction.

```
function main():
  push(($q_0, Z_0, \alpha_0$), stack)
  while (stack $\neq \emptyset$) do
    ($q, Z, \alpha$) := top(stack); pop(stack)
    explore($q, Z, \alpha$)
    resolve()
  return "empty"

function explore($q, Z, \alpha$):
  if ($q$ is accepting)
    exit "not empty"
  if ($\exists$ ($q, Z', \alpha'$) nontentative
            and s.t. $Z \subseteq Closure_{\alpha'}(Z')$)
    mark ($q, Z, \alpha$) tentative wrt ($q, Z', \alpha'$)
    $\alpha := \alpha'$; propagate($parent(q, Z, \alpha)$)
  else
    propagate($q, Z, \alpha$)
    for each ($q_s, Z_s, \alpha_s$) in $children(q, Z, \alpha)$ do
      if ($Z_s \neq \emptyset$)
        explore($q_s, Z_s, \alpha_s$)

function resolve():
  for each ($q, Z, \alpha$) tentative wrt ($q, Z', \alpha'$) do
    if ($Z \nsubseteq Closure_{\alpha'}(Z')$)
      mark ($q, Z, \alpha$) nontentative
      $\alpha := -\infty$; propagate($parent(q, Z, \alpha)$)
      push(($q, Z, \alpha$), stack)

function propagate($q, Z, \alpha$):
  $\alpha := max_{(q,Z,\alpha) \xrightarrow{g;R} (q',Z',\alpha')}$ maxedge($g, R, \alpha'$)
  if ($\alpha$ has changed)
    for each ($q_t, Z_t, \alpha_t$) tentative wrt ($q, Z, \alpha$) do
      $\alpha_t := \alpha$; propagate($parent(q_t, Z_t, \alpha_t)$)
    if (($q, Z, \alpha$) $\neq$ ($q_0, Z_0, \alpha_0$))
      propagate($parent(q, Z, \alpha)$)

function maxedge($g, R, \alpha$):
  let $\alpha_R = \lambda x.$ if $x \in R$ then $-\infty$ else $\alpha(x)$
  let $\alpha_g = \lambda x.$ if $x \# c$ in $g$ then $c$ else $-\infty$
  return ($\lambda x.\ max(\alpha_R(x), \alpha_g(x))$)
```

coming from unreachable transitions. We comment more on advantages of this approach in Section 5.

Our modified algorithm is given in Figure 1. It computes a tree whose nodes are triples $(q, Z, \alpha)$ where $(q, Z)$ is a node of $ZG(\mathcal{A})$ and $\alpha$ is a bound function. Each node $(q, Z, \alpha)$ has as many child nodes $(q_s, Z_s, \alpha_s)$ as there are successors $(q_s, Z_s)$ of $(q, Z)$ in $ZG(\mathcal{A})$. Notice that this includes successors with an empty zone $Z_s$, which are however not further unfolded. These nodes must be included for correctness of our constant propagation procedure. By default bound functions map each clock to $-\infty$. They are later updated as explained below. Each node is further marked either *tentative* or *nontentative*. The leaf nodes $(q, Z, \alpha)$ of the tree are either deadlock nodes (either there is no transition out of state $q$ or $Z$ is empty), or *tentative* nodes. All the other nodes are marked *nontentative*.

Our algorithm starts from the root node $(q_0, Z_0, \alpha_0)$, consisting of the initial state, initial zone, and the function mapping each clock to $-\infty$. It repeatedly alternates an exploration and a resolution phase as described below.

## Exploration phase

Before exploring a node $n = (q, Z, \alpha)$ the function `explore` checks if $q$ is accepting and $Z$ is not empty; if it is so then $\mathcal{A}$ has an accepting run. Otherwise the algorithm checks if there exists a *nontentative* node $n' = (q', Z', \alpha')$ in the current tree such that $q = q'$ and $Z \subseteq Closure_{\alpha'}(Z')$. If yes, $n$ becomes a *tentative* node and its exploration is temporarily stopped as each state reachable from $n$ is also reachable from $n'$. If none of these holds, the successors of the node are explored. The exploration terminates since $Closure_{\alpha}$ has a finite range.

When the exploration algorithm gets to a new node, it propagates the bounds from this node to all its predecessors. The goal of these propagations is to maintain the following invariant. For every node $n = (q, Z, \alpha)$:

1. if $n$ is *nontentative*, then $\alpha$ is the maximum of the $\alpha_s$ from all successor nodes $(q_s, Z_s, \alpha_s)$ of $n$ (taking into account guards and resets as made precise in the function `maxedge`);
2. if $n$ is *tentative* with respect to $(q', Z', \alpha')$, then $\alpha$ is equal to $\alpha'$.

The result of propagation is analogous to the inequalities seen in the static guard analysis [2], however now applied to the zone graph, on-the-fly. Hence, the bounds associated to each node $(q, Z, \alpha)$ never exceed those that are computed by the static guard analysis.

A delicate point about this procedure is handling of tentative nodes. When a node $n$ is marked *tentative*, we have $\alpha = \alpha'$. However the value of $\alpha'$ may be updated when the tree is further explored. Thus each time we update the bounds function of a node, it is not only propagated upward in the tree but also to the nodes that are tentative with respect to $n'$.

This algorithm terminates as the bound functions in each node never decrease and are bounded. From the invariants above, we get that in every node, $\alpha$ is a solution to the equations in [2] applied on $ZG(\mathcal{A})$.

It could seem that the algorithm will be forced to do a high number of propagations of bounds. The experiments reported in Section 5 show that the present very simple approach to bound propagation is good enough. Since we propagate the bounds as soon as they are modified, most of the time, the value of $\alpha$ does not change in line 30 of function `propagate`. In general, bounds are only propagated on very short distances in the tree, mostly along one single edge. For this reason we do not concentrate on optimizing the function `propagate`. In the implementation we use the presented function augmented with a minor "optimization" that avoids calculating maximum over all successors in line 30 when it is not needed.

### Resolution phase

Finally, as the bounds may have changed since $n$ has been marked tentative, the function `resolve` checks for the consistency of *tentative* nodes. If $Z \subseteq \mathit{Closure}_{\alpha'}(Z')$ is not true anymore, $n$ needs to be explored. Hence it is viewed as a new node: the bounds are set to $-\infty$ and $n$ is pushed on the *stack* for further consideration in the function `main`. Setting $\alpha$ to $-\infty$ is safe as $\alpha$ will be computed and propagated when $n$ is explored. We perform also a small optimization and propagate this bound upward, thereby making some bounds decrease.

The resolution phase may provide new nodes to be explored. The algorithm terminates when this is not the case, that is when all tentative nodes remain tentative. We can then conclude that no accepting state is reachable.

▶ **Theorem 7.** *An accepting state is reachable in $ZG(\mathcal{A})$ iff the algorithm reaches a node with an accepting state and a non-empty zone.*

## 4.3 Handling LU approximations

Recall that $\mathit{Extra}^+_{LU}(Z)$ approximation used two bounds: $L_x$ and $U_x$ for each clock $x$. In our algorithm we can easily propagate LU bounds instead of just maximal bounds. We can also replace the test $Z \subseteq \mathit{Closure}_{\alpha'}(Z')$ by $Z \subseteq \mathit{Closure}_{\alpha'}(\mathit{Extra}^+_{L'U'}(Z'))$, where $L'$ and $U'$ are the bounds calculated for $(q', Z')$ and $\alpha'_x = \max(L'_x, U'_x)$ for every clock $x$. As discussed in Section 3.3, this test can be done efficiently too. The proof of correctness of the resulting algorithm is only slightly more complicated.

## 5 Experimental results

We have implemented the algorithm from Figure 1, and have tested it on classical benchmarks. The results are presented in Table 1, along with a comparison to UPPAAL and our implementation of UPPAAL's core algorithm that uses the $\mathit{Extra}^+_{LU}$ extrapolation [3] and computes bounds by static analysis [2]. Since we have not considered symmetry reduction [12] in our tool, we have not used it in UPPAAL either.

**Table 1** Experimental results: number of visited nodes and running time with a timeout (T.O.) of 60 seconds. Experiments done on a MacBook with 2.4GHz Intel Core Duo processor and 2GB of memory running MacOS X 10.6.7.

| Model | Our algorithm | | UPPAAL's algorithm | | UPPAAL 4.1.3 (-n4 -C -o1) | |
|---|---|---|---|---|---|---|
| | nodes | s. | nodes | s. | nodes | s. |
| $\mathcal{A}_1$ | 2 | 0.00 | 10003 | 0.07 | 10003 | 0.07 |
| $\mathcal{A}_2$ | 7 | 0.00 | 3999 | 0.60 | 2003 | 0.01 |
| $\mathcal{A}_3$ | 3 | 0.00 | 10004 | 0.37 | 10004 | 0.32 |
| CSMA/CD7 | 5031 | 0.32 | 5923 | 0.27 | – | T.O. |
| CSMA/CD8 | 16588 | 1.36 | 19017 | 1.08 | – | T.O. |
| CSMA/CD9 | 54439 | 6.01 | 60783 | 4.19 | – | T.O. |
| FDDI10 | 459 | 0.02 | 525 | 0.06 | 12049 | 2.43 |
| FDDI20 | 1719 | 0.29 | 2045 | 0.78 | – | T.O. |
| FDDI30 | 3779 | 1.29 | 4565 | 4.50 | – | T.O. |
| Fischer7 | 7737 | 0.42 | 20021 | 0.53 | 18374 | 0.35 |
| Fischer8 | 25080 | 1.55 | 91506 | 2.48 | 85438 | 1.53 |
| Fischer9 | 81035 | 5.90 | 420627 | 12.54 | 398685 | 8.95 |
| Fischer10 | – | T.O. | – | T.O. | 1827009 | 53.44 |

The comparison to UPPAAL is not meaningful for the CSMA/CD and the FDDI protocols. Indeed, UPPAAL runs out of time even if we significantly increase the time allowed; switching to breadth-first search has not helped either. We suspect that this is due to the order in which UPPAAL takes the transitions in the automaton. For this reason in columns 4 and 5, we provide results from our own implementation of UPPAAL's algorithm that takes transitions in the same order as the implementation of our algorithm. Although RED also uses approximations, it is even more difficult to draw a meaningful comparison with it, since it uses symbolic state representation unlike UPPAAL or our tool. Since this paper is about approximation methods, and not tool comparison, we leave more extensive comparisons as further work.

The results show that our algorithm provides important gains. Analyzing the results more closely we could see that both the use of closure, and on-the-fly computation of bounds are important. In Fischer's protocol our algorithm visits much less nodes. In the FDDI protocol with $n$ processes, the DBMs are rather big square matrices of order $3n+2$. Nevertheless our inclusion test based on *Closure* is significantly better in the running time. The CSMA/CD case shows that the cost of bounds propagation does not always counterbalance the gains. However the overhead is not very high either. We comment further on the results below.

The first improvement comes from the computation of the maximal bounds used for the abstraction as demonstrated by the examples $\mathcal{A}_2$ (Figure 2), Fischer and CSMA/CD that correspond to three different situations. In the $\mathcal{A}_2$ example, the transition that yields the big bound $10^4$ on $y$ in $q_0$ is not reachable from any $(q_0, Z)$, hence we just get the lower bound 20 on $y$ in $(q_0, Z)$, and a subsequent gain in performance.

The automaton $\mathcal{A}_1$ in Figure 2 illustrates the gain on the CSMA/CD protocol. The transition from $q_0$ to $q_1$ is disabled as it must synchronize on letter $a!$. The static analysis algorithm [2] ignores this fact, hence it associates bound $10^4$ to $y$ in $q_0$. Since our algorithm computes the bounds on-the-fly, $y$ is associated the bound 10 in every node $(q_0, Z)$. We observe that UPPAAL's algorithm visits 10003 nodes on $ZG(\mathcal{A}_1)$ whereas our algorithm only visits 2 nodes. The same situation occurs in the CSMA/CD example. However despite the improvement in the number of nodes (roughly 10%) the cost of computing the bounds impacts the running time negatively.

The gains that we observe in the analysis of the Fischer's protocol are explained by the automaton $\mathcal{A}_3$ in Figure 2. $\mathcal{A}_3$ has a bounded integer variable $n$ that is initialized to 0.

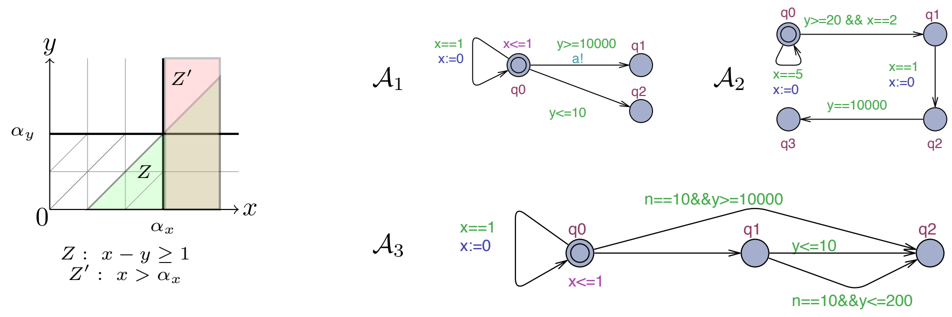

**Figure 2** Examples explaining gains obtained with the algorithm.

Hence, the transitions from $q_0$ to $q_2$, and from $q_1$ to $q_2$, that check if $n$ is equal to 10 are disabled. This is ignored by the static analysis algorithm that associates the bound $10^4$ to clock $y$ in $q_0$. Our algorithm however associates the bound 10 to $y$ in every node $(q_0, Z)$. We observe that UPPAAL's algorithm visits 10004 nodes whereas our algorithm only visits 3 nodes. A similar situation occurs in the Fischer's protocol. We include the last row to underline that our implementation is not as mature as UPPAAL. We strongly think that UPPAAL could benefit from methods presented here.

The second kind of improvement comes from the $\mathit{Closure}_\alpha$ abstraction that particularly improves the analysis of the Fischer's and the FDDI protocols. The situation observed on the FDDI protocol is explained in Figure 2. For the zone $Z$ in the figure, by definition $\mathit{Extra}^+_{LU}(Z) = Z$, and in consequence $Z' \not\subseteq Z$. However, $Z' \subseteq \mathit{Closure}_\alpha(Z)$. On FDDI and Fischer's protocols, our algorithm performs better due to the non-convex approximation.

## 6 Conclusions

We have proposed a new algorithm for checking reachability properties of timed automata. The algorithm has two sources of improvement that are quite independent: the use of the $\mathit{Closure}_\alpha$ operator, and the computation of bound functions on-the-fly.

Apart from immediate gains presented in Table 1, we think that our approach opens some new perspectives on analysis of timed systems. We show that the use of non-convex approximations can be efficient. We have used very simple approximations, but it may be well the case that there are more sophisticated approximations to be discovered. The structure of our algorithm permits to calculate bounding constants on the fly. One should note that standard benchmarks are very well understood and very well modeled. In particular they have no "superfluous" constraints or clocks. However in not-so-clean models coming from systems in practice one can expect the on-the-fly approach to be even more beneficial.

There are numerous directions for further research. One of them is to find other approximation operators. Methods for constraint propagation also deserve a closer look. We believe that our approximations methods are compatible with partial order reductions [12, 14]. We hope that the two techniques can benefit from each other.

## References

1 R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

2 G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In *TACAS*, volume 2619 of *LNCS*, pages 254–270. Springer, 2003.

**3** G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelanek. Lower and upper bounds in zone-based abstractions of timed automata. *Int. Journal on Software Tools for Technology Transfer*, 8(3):204–215, 2006.

**4** G. Behrmann, A. David, K. G Larsen, J. Haakansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *QEST'06*, pages 125–126, 2006.

**5** J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004.

**6** B. Bérard, B. Bouyer, and A. Petit. Analysing the PGM protocol with UPPAAL. *Int. Journal of Production Research*, 42(14):2773–2791, 2004.

**7** P. Bouyer. Forward analysis of updatable timed automata. *Form. Methods in Syst. Des.*, 24(3):281–320, 2004.

**8** C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Form. Methods Syst. Des.*, 1(4):385–415, 1992.

**9** C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS'98*, volume 1384 of *LNCS*, pages 313–329. Springer, 1998.

**10** D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *AVMFSS*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

**11** K. Havelund, A. Skou, K. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *RTSS*, pages 2–13, 1997.

**12** M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. Vaandrager. Adding symmetry reduction to UPPAAL. In *Int. Workshop on Formal Modeling and Analysis of Timed Systems*, volume 2791 of *LNCS*, pages 46–59. Springer, 2004.

**13** F. Herbreteau, D. Kini, B. Srivathsan, and I. Walukiewicz. Using non-convex approximations for efficient analysis of timed automata. http://hal.archives-ouvertes.fr/inria-00559902/en/, 2011. Extended version with proofs.

**14** J. Malinowski and P. Niebert. SAT based bounded model checking with partial order semantics for timed automata. In *TACAS*, volume 6015 of *LNCS*, pages 405–419, 2010.

**15** G. Morbé, F. Pigorsch, and C. Scholl. Fully symbolic model checking for timed automata. In *CAV'11*, volume 6806 of *LNCS*, pages 616–632. Springer, 2011.

**16** Farn Wang. Efficient verification of timed automata with BDD-like data structures. *Int. J. on Software Tools for Technology Transfer*, 6:77–97, 2004.