Lazy Reachability Checking for Timed Automata using Interpolants

Tamás Tóth[*] and István Majzik

Budapest University of Technology and Economics
Department of Measurement and Information Systems
{totht,majzik}@mit.bme.hu

Abstract. To solve the reachability problem for timed automata, model checkers usually apply forward search and zone abstraction. To ensure efficiency and termination, the computed zones are generalized using maximal constants obtained from guards either by static analysis or lazily for a given path. In this paper, we propose a lazy method based on zone abstraction that, instead of the constants in guards, considers the constraints themselves. The method is a combination of forward search, backward search and interpolation over zones: if the zone abstraction is too coarse, we propagate a zone representing bad states backwards using backward search, and use interpolation to extract a relevant zone to strengthen the current abstraction. We propose two refinement strategies in this framework, and evaluate our method on the usual benchmark models for timed automata. Our experiments show that the proposed method compares favorably to known methods based on efficient lazy non-convex abstractions.

Keywords: timed automata, model checking, reachability, zone abstraction, interpolation

1 Introduction

Timed automata [1] is a widely used formalism for the modeling and verification of real-time systems. The reachability problem deals with the question whether a given error state is reachable from an initial state along the transitions of the automaton. The standard solution of this problem involves performing a forward exploration in the so-called zone-graph induced by the automaton [9].

To ensure performance and termination, model checkers for timed automata usually apply some sort of generalization of zones based on maximal lower- and upper bounds [3] (*LU*-bounds) appearing in the guards of the automaton. This can be performed directly by extrapolation [3] parametrized by bounds obtained by static analysis [2]. Alternatively, bounds can be propagated lazily for all transitions [12] or along an infeasible path [11], which, combined with

[*] This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

Authors' manuscript. Published in A. Abate, G. Geeraerts (eds.): *Formal Modelling and Analysis of Timed Systems*, LNCS 10419, 2017. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-65765-3_15.

an efficient method for inclusion checking [13] with respect to a non-convex abstraction induced by the bounds, results in an efficient method for reachability checking of timed automata. This latter approach can be seen as a variant of counterexample-guided abstraction refinement [8] (CEGAR), a technique widely used in model checking.

In this paper, we propose a similar lazy algorithm for reachability checking of timed automata. However, instead of propagating the bounds appearing in guards, the algorithm considers the guards themselves. If the abstraction is too coarse to exclude an infeasible path, a zone representing the guards of a disabled transition is propagated backwards using pre-image computation. Based on the pre-image, we compute a zone strong enough to block the disabled transition in form of an interpolant [14]. In a similar fashion, we use interpolation to effectively prune the search space by enforcing coverage of a newly discovered state with an already visited state when possible. We propose two refinement strategies in this framework. Both methods are a combination of forward search, backward search and zone interpolation, and can be considered as a generalization of zone interpolation to sequences of transitions of a timed automaton.

We compared the proposed interpolation based method and the non-convex *LU*-abstraction based method [11] on the usual benchmark models for timed automata. Results show that our method performs similarly to the highly sophisticated algorithm of [11], and in cases can even generate a smaller state space. Moreover, it turned out that for some models the proposed refinement strategies are less sensitive to search order, thus are more robust against bad decisions during search.

Comparison to related work. Lazy abstraction [10] is an approach widely used for model checking, and in particular for model checking software. It consists of building an abstract reachability graph on-the fly, representing an abstraction of the system, and refining a part of the tree in case a spurious counterexample is found. Lazy abstraction with interpolants [15] (also known as Impact) and lazy annotation [16] are both lazy abstraction techniques for software where refinement is performed using interpolant generation.

For timed automata, a lazy abstraction approach based on non-convex *LU*-abstraction and on-the-fly propagation of bounds has been proposed [11]. A significant difference of this algorithm compared to usual lazy abstraction algorithms is that it builds an abstract reachability graph that preserves exact reachability information (a so-called adaptive simulation graph). As a consequence it is able to apply refinement as soon as the abstraction admits a transition disabled in the concrete system. In our work, we apply the same approach, but for a different abstract domain, with different refinement strategies.

The work closest to ours is difference bound constraint abstraction [18]. The refinement method presented there and our refinement strategy we refer to as the binary (bin) strategy are highly analogous, and both are very similar to lazy annotation. However, our refinement strategy that we refer to as the sequence (seq) strategy is different in concept. Moreover, in [18], abstractions are sets of difference constraints, and refinement rules are defined on a case-by-case basis for guards, resets and delay. In our paper, we represent abstractions as canonical difference bound matrices, and define abstraction refinement in more general terms, as a combination of symbolic forward and backward search and zone interpolation. This formulation enables a simple generalization of our approach to automata with diagonal constraints in guards [6] and to updatable timed automata [5], as well as to the application of backward exploration. Moreover, by representing abstractions as canonical difference bound matrices, known zone-based abstraction methods can be considered orthogonal to our approach.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we define the notations used throughout the paper, and present the theoretical background of our work. In Section 3 we propose a lazy reachability checking algorithm based on zone abstraction for timed automata. We propose two methods for abstraction refinement in Section 4. Section 5 describes experiments performed on the proposed algorithm. Finally, conclusions are given in Section 6.

2 Background and Notations

Let X be a set of *clock variables* over $\mathbb{R}$. We assume $x_0 \in X$, where x_0 is a distinguished reference clock with constant value 0. A *clock constraint* over X is a conjunction of atoms of the form $x_i - x_j \prec c$ where $x_i, x_j \in X$, $c \in \mathbb{Z}$ and $\prec\ \in \{<, \leq\}$. We denote the set of clock constraints over X by $\Phi(X)$.

A *clock valuation* over X is a function $\eta : X \to \mathbb{R}$. We denote by $\mathit{Eval}(X)$ the set of clock valuations over X, and by $\mathbf{0} \in \mathit{Eval}(X)$ the clock valuation where $\mathbf{0}(x) = 0$ for all $x \in X$. For a real number $\delta \geq 0$ and for all $x \in X$, let $(\eta + \delta)(x) = \eta(x) + \delta$. Moreover, for $R \subseteq X$ and for all $x \in X$, let $([R]\,\eta)(x) = 0$ if $x \in R$ and $([R]\,\eta)(x) = \eta(x)$ otherwise. For a clock constraint $\varphi \in \Phi(X)$, we denote by $\eta \models \varphi$ iff φ is satisfied under valuation η. Furthermore, let $[\![\varphi]\!] = \{\eta \mid \eta \models \varphi\}$.

2.1 Timed automata

Definition 1 (Timed automaton). *Syntactically, a timed automaton is a tuple $\mathcal{A} = (L, X, T, \ell_0)$ where*

- *L is a finite set of locations,*
- *X is a finite set of clock variables,*
- *$T \subseteq L \times \Phi(X) \times \mathcal{P}(X) \times L$ is a finite set of transitions where for a transition $(\ell, g, R, \ell') \in T$, constraint g is a guard and R is a set containing clocks to be reset, and*
- *$\ell_0 \in L$ is the initial location.*

A state of $\mathcal{A}$ is a pair (ℓ, η) where $\ell \in L$ and $\eta \in \mathit{Eval}(X)$.

Definition 2 (Semantics). *The operational semantics of a timed automaton is given by a labeled transition system with initial state* $(\ell_0, \mathbf{0})$ *and two kinds of transitions:*

- Delay*:* $(\ell, \eta) \xrightarrow{\delta} (\ell, \eta + \delta)$ *for some* $\delta \geq 0$*;*
- Action*:* $(\ell, \eta) \xrightarrow{t} (\ell', [R]\,\eta)$ *for some transition* $t = (\ell, g, R, \ell')$ *where* $\eta \models g$*.*

A *run* of a timed automaton is a sequence of states from the initial state along the transition relation $(\ell_0, \eta_0) \xrightarrow{\alpha_1} (\ell_1, \eta_1) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} (\ell_n, \eta_n)$ where $\eta_0 = \mathbf{0}$ and $\alpha_i \in T \cup \mathbb{R}_{\geq 0}$ for all $0 \leq i \leq n$. A location $\ell \in L$ is *reachable* iff there exists a run such that $\ell_n = \ell$.

2.2 Symbolic semantics

As the concrete semantics of a timed automaton is infinite due to real valued clock variables, model checkers are often based on a symbolic semantics defined in terms of zones. A *zone* $Z \in \mathcal{Z}$ is the solution set of a clock constraint $\varphi \in \Phi(X)$, that is $Z = [\![\varphi]\!]$. For zones Z and Z', we will denote by $Z \sqsubseteq Z'$ iff $Z \subseteq Z'$. Moreover, if Z and Z' are zones and $t \in T$, then

- $\bot = \emptyset$,
- $\top = \mathit{Eval}(X)$,
- $Z \sqcap Z' = Z \cap Z'$,
- $Z_0 = \{\eta \mid \eta = \mathbf{0} + \delta \text{ for some } \delta \geq 0\}$,
- $\mathbf{post}_t(Z) = \left\{\eta' \mid (\ell, \eta) \xrightarrow{t} s \xrightarrow{\delta} (\ell', \eta') \text{ for some } \eta \in Z \text{ and } \delta \geq 0\right\}$, and
- $\mathbf{pre}_t(Z') = \left\{\eta \mid (\ell, \eta) \xrightarrow{t} s \xrightarrow{\delta} (\ell', \eta') \text{ for some } \eta' \in Z' \text{ and } \delta \geq 0\right\}$

are also zones. Zones are not closed under complementation, but the complement of any zone is the union of finitely many zones. For a zone Z, we are going to denote a finite set of such zones by $\neg Z$.

The functions $\mathbf{post}_t(Z)$ and $\mathbf{pre}_t(Z)$ represent the strongest postcondition and weakest precondition of Z with respect to a transition t of a timed automaton, repsectively. We are going to use the following simple lemma.

Lemma 1. *Let* A*,* B *be zones and* $t \in T$ *a transition. Then* $A \sqcap \mathbf{pre}_t(B) \sqsubseteq \bot$ *iff* $\mathbf{post}_t(A) \sqcap B \sqsubseteq \bot$*.*

Using **post**, we can define a zone-based symbolic semantics for timed automata.

Definition 3 (Symbolic semantics). *The symbolic semantics of a timed automaton is given by a labeled transition system with states of the form* (ℓ, Z)*, with initial state* (ℓ_0, Z_0)*, and with transitions of the form* $(\ell, Z) \overset{t}{\Rightarrow} (\ell', \mathbf{post}_t(Z))$ *where* $t = (\ell, g, R, \ell')$*.*

Definition 4 (Symbolic run). *A symbolic run of a timed automaton is a sequence* $(\ell_0, Z_0) \overset{t_1}{\Rightarrow} (\ell_1, Z_1) \overset{t_2}{\Rightarrow} \ldots \overset{t_n}{\Rightarrow} (\ell_n, Z_n)$ *where* $Z_n \neq \bot$*.*

Proposition 1. *For a timed automaton, a location* $\ell \in L$ *is reachable iff there exists a symbolic run with* $\ell_n = \ell$*.*

2.3 Difference Bound Matrices

Clock constraints and thus zones can be efficiently represented by difference bound matrices.

A *bound* is either ∞, or a finite bound of the form $(m, \prec)$ where $m \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. Difference bounds can be totally ordered by "strength", that is, $(m, \prec) < \infty$, $(m_1, \prec_1) < (m_2, \prec_2)$ iff $m_1 < m_2$ and $(m, <) < (m, \leq)$. Moreover the sum of two bounds is defined as $b+\infty = \infty$, $(m_1, \leq)+(m_2, \leq) = (m_1+m_2, \leq)$ and $(m_1, <) + (m_2, \prec) = (m_1 + m_2, <)$.

A *difference bound matrix* (DBM) over $X = \{x_0, x_1, \ldots, x_n\}$ is a square matrix D of bounds of order $n + 1$ where an element $D_{ij} = (m, \prec)$ represents the clock constraint $x_i - x_j \prec m$. We denote by $[\![D]\!]$ the zone induced by the conjunction of constraints stored in D. We say that D is *consistent* iff $[\![D]\!] \neq \emptyset$. The following is a simple sufficient and necessary condition for a DBM to be inconsistent.

Proposition 2. *A DBM D is inconsistent iff there exists a negative cycle in D, that is, a set of pairs of indexes $\{(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)\}$ such that $D_{i_1,i_2} + \ldots + D_{i_{k-1},i_k} + D_{i_k,i_1} < (0, \leq)$.*

For a consistent DBM D, we say it is *canonical* iff constraints in it can not be strengthened without losing solutions, formally, iff $D_{ii} = (0, \leq)$ for all $0 \leq i \leq n$ and $D_{ij} \leq D_{ik} + D_{kj}$ for all $0 \leq i, j, k \leq n$. For convenience, we will also consider the inconsistent DBM D with the single finite bound $D_{00} = (0, <)$ canonical. Up to the ordering of clocks, the canonical form is unique. Moreover, the zone operations in Section 2.2 can be efficiently implemented over canonical DBMs [4]. Therefore, we will refer to a canonical DBM D (syntax) and the zone $[\![D]\!]$ it represents (semantics) interchangeably throughout the paper.

For two DBMs A and B, we will denote by $\min(A, B)$ the (not necessarily canonical) DBM D where $D_{ij} = \min(A_{ij}, B_{ij})$, which encodes $[\![A]\!] \cap [\![B]\!]$.

3 Algorithm

In this section, we present our algorithm for lazy reachability checking of timed automata.

3.1 Adaptive simulation graph

The definitions and propositions presented here are adaptations of concepts introduced in [11] to our convex, zone-based setting.

Definition 5 (Unwinding). *An unwinding of a timed automaton (L, X, T, ℓ_0) is a tuple $U = (V, E, v_0, M_v, M_e, \rhd)$ where*

- *(V, E) is a directed tree rooted at node $v_0 \in V$,*
- *$M_v : V \to L$ is the vertex labeling,*

- *$M_e : E \to T$ is the edge labeling, and*
- *$\rhd \subseteq V \times V$ is the (functional) covering relation.*

For an unwinding we require that the following properties hold:

- *$M_v(v_0) = \ell_0$,*
- *for each edge $(v, v') \in E$ the transition $M_e(v, v') = (\ell, g, R, \ell')$ is such that $M_v(v) = \ell$ and $M_v(v') = \ell'$,*
- *for all v and v' such that $v \rhd v'$ it holds that $M_v(v) = M_v(v')$.*

Informally, the purpose of the covering relation $\rhd$ is to mark if the search space has been pruned at a node due to an other node that admits all runs possible from the covered node. For convenience, we define the following shorthand notations: $\ell_v = M_v(v)$ and $t_{v,v'} = M_e(v, v')$.

Definition 6 (Adaptive simulation graph). *An adaptive simulation graph (ASG) for a timed automaton $\mathcal{A}$ is a tuple $G = (U, \psi_Z, \psi_W)$ where*

- *U is an unwinding of $\mathcal{A}$, and*
- *$\psi_Z, \psi_W : V \to \mathcal{Z}$ are labelings of vertices by zones.*

We will use the following shorthand notations: $Z_v = \psi_Z(v)$ and $W_v = \psi_W(v)$. Later, we will ensure that Z_v represents the exact set of reachable valuations for v, and W_v an overapproximation of it.

A node v is *expanded* iff it has a successor for all transitions $t = (\ell, g, R, \ell')$ such that $\ell_v = \ell$. Without loss of generality, we assume that for each location the automaton has at least one outgoing transition, thus if a node is expanded, then it is not a leaf. A node v is *feasible* iff $W_v \neq \bot$. It is covered iff $v \rhd v'$ for some node v'. It is *excluded* iff it is covered, infeasible or it has an excluded parent. A node is *complete* iff it is either expanded or excluded. A node is ℓ-safe iff $\ell_v \neq \ell$.

For an ASG to be useful for reachability checking, we have to introduce restrictions on the labelings ψ_Z and ψ_W.

Definition 7 (Well-labeled node). *A node v of an ASG G for a timed automaton $\mathcal{A}$ is well-labeled iff the following conditions hold:*

- *(initiation) if $v = v_0$, then (a) $Z_v = Z_0$ and (b) $Z_0 \sqsubseteq W_v$;*
- *(consecution) if $v \neq v_0$, then for its parent u and the transition $t = t_{u,v}$ we have (a) $Z_v = \mathbf{post}_t(Z_u)$ and (b) $\mathbf{post}_t(W_u) \sqsubseteq W_v$;*
- *(coverage) if $v \rhd v'$ for some node v', then $W_v \sqsubseteq W_{v'}$, and v' is not excluded;*
- *(simulation) if $Z_v = \bot$, then $W_v = \bot$.*

The above definitions for nodes can be extended to ASGs: an ASG is complete, ℓ-safe or well-labeled iff all its nodes are complete, ℓ-safe or well-labeled, respectively. As the conditions for well-labeledness suggest, the main challenge for the construction of a well-labeled ASG is how the labeling ψ_W is computed. In Section 4, we propose two strategies for computing a labeling that satisfies well-labeledness. A well labeled ASG preserves reachability information, which is expressed by the following proposition.

Proposition 3. *Let G be a complete, well-labeled ASG for a timed automaton $\mathcal{A}$. Then $\mathcal{A}$ has a symbolic run $(\ell_0, Z_0) \overset{t_1}{\Rightarrow} (\ell_1, Z_1) \overset{t_2}{\Rightarrow} \dots \overset{t_n}{\Rightarrow} (\ell_n, Z_n)$ iff G has a non-excluded node v such that $\ell_v = \ell_n$.*

Proof. The left-to right direction is a consequence of Lemma 2, and the converse is a consequence of Lemma 3. □

Lemma 2. *Let G be a complete, well-labeled ASG for a timed automaton $\mathcal{A}$. If $\mathcal{A}$ has a symbolic run $(\ell_0, Z_0) \overset{t_1}{\Rightarrow} (\ell_1, Z_1) \overset{t_2}{\Rightarrow} \dots \xRightarrow{t_{n-1}} (\ell_{n-1}, Z_{n-1}) \overset{t_n}{\Rightarrow} (\ell, Z)$ then G has a non-excluded node v such that $\ell = \ell_v$ and $Z \sqsubseteq W_v$.*

Proof. We prove the statement by induction on the length n of the symbolic run. If $n = 0$, then $\ell = \ell_0$ and $Z = Z_0$, thus v_0 is a suitable witness by condition *initiation(b)*. Suppose the statement holds for runs of length at most $n - 1$. Thus there exists a non-excluded node v_{n-1} such that $\ell_{n-1} = \ell_{v_{n-1}}$ and $Z_{n-1} \sqsubseteq W_{v_{n-1}}$. As v_{n-1} is complete and not excluded, it is expanded, thus by condition *consecution(b)*, there is a successor node v_n for transition t_n such that $\ell_n = \ell_{v_n}$ and $\mathbf{post}_{t_n}(W_{n-1}) \sqsubseteq W_{v_n}$. Clearly, $Z \sqsubseteq W_n$, as $Z = \mathbf{post}_{t_n}(Z_{n-1})$ and $\mathbf{post}_t$ is monotonic for any $t \in T$. Thus if v_n is not covered then it is a suitable witness. Otherwise there exists a node $v \in V$ such that $v_n \rhd v$. By condition *coverage*, we know that $W_{v_n} \sqsubseteq W_v$ and v is not excluded, thus it is a suitable witness. □

Lemma 3. *Let G be an ASG for a timed automaton $\mathcal{A}$. Let v be a non-excluded, well-labeled node of G such that all its ancestors are well-labeled. Then $\mathcal{A}$ has a symbolic run $(\ell_0, Z_0) \overset{t_1}{\Rightarrow} (\ell_1, Z_1) \overset{t_2}{\Rightarrow} \dots \overset{t}{\Rightarrow} (\ell_v, Z_v)$.*

Proof. We prove the statement by induction on the depth n of v in the tree. If $n = 0$, then $v = v_0$. Thus $\ell_v = \ell_0$ and $Z_v = Z_0$ by condition *initiation(a)*, and (ℓ_0, Z_0) is a suitable run of $\mathcal{A}$. Assume that the statement holds for nodes in depth at most $n-1$. Let u be the parent of v. As u is non-excluded, well-labeled, and all its ancestors are well-labeled, there exists a symbolic run to (ℓ_u, Z_u). By condition *consecution(a)*, we have $\mathbf{post}_t(Z_u) = Z_v$ for $t = t_{u,v}$. As v is not excluded, $Z_v \neq \bot$ by condition *simulation*, thus the run to u can be extended to a run to v by appending to it (ℓ_v, Z_v) for t. □

Remark 1. Note that for an automaton $\mathcal{A}$, the labeling ψ_W can be chosen so that the ASG is finite. A way to construct such an ASG is for example by taking $W_v = \mathit{Extra}^+_{LU}(Z_v)$ [3] for some bound functions L and U statically computed for ℓ_v, for all nodes v. Similarly, the termination of any reasonable algorithm for constructing a well-labeled ASG can be ensured by maintaining the additional invariant $W_v = \mathit{Extra}^+_{LU}(W_v)$ for all nodes v. As doing so is straightforward, termination can be considered an issue orthogonal to abstraction computation. In this paper, we focus on the latter.

3.2 Algorithm

The pseudocode of the reachability algorithm is shown in Algorithm 1. The main procedure of the algorithm is EXPLORE, which gets as input a timed automaton $\mathcal{A}$ and an error location $\ell_e \in L$. Upon termination, it either witnesses reachability by a symbolic run of $\mathcal{A}$ to ℓ_e, or proves unreachability of ℓ_e for $\mathcal{A}$ with a well-labeled, complete, ℓ_e-safe ASG.

The main data structures of the algorithm are the ASG G over set of nodes V, and sets *waiting* and *passed*, both of which store nodes from V. Informally, *waiting* stores leaves that are not yet excluded, and *passed* stores nodes that have been expanded. The algorithm consists of three subprocedures, EXPAND, COVER, and REFINE. The procedure COVER attempts to add a covering edge for a node. Procedure EXPAND creates the successors for a node. For a node v and zone W such that $Z_v \sqsubseteq W$, procedure REFINE enforces that also $W_v \sqsubseteq W$ holds. This is performed by calls to a procedure BLOCK, for which two possible algorithms based on interpolation are given in Section 4. The contract of BLOCK asserts that whenever zones Z_v and B are inconsistent, then after the call, the inconsistency of W_v and B is also ensured. Note that this condition is sufficient to satisfy the contract of REFINE.

Informally, the algorithm employs the following strategy. The algorithm consists of the single loop in line 10 that consumes nodes from *waiting* one by one. If *waiting* becomes empty, then $\mathcal{A}$ is deemed safe. Otherwise, a node v is removed from *waiting*. If $Z_v \sqsubseteq \bot$, then *simulation* is established by calling to REFINE. Otherwise, if the node represents an error location, then $\mathcal{A}$ is deemed unsafe. Otherwise, in order to avoid unnecessary expansion of the node, the algorithm tries to cover it. This is attempted by a call to COVER to enforce *coverage* by a candidate node v'. As the labeling of v' might be strengthened during the call as a side effect, after the call, the condition for coverage is checked. If it is satisfied, v gets covered. Otherwise, v is put back to *waiting*. If there are no suitable candidates for coverage, then the algorithm expands the node by a call to EXPAND, puts it in *passed*, and puts all its newly created successors in *waiting*.

To show correctness of EXPLORE w. r. t. the annotation specified in line 1, we will refer to the following subsets of V: let *infeasible* $= \{v \mid v \text{ is infeasible}\}$ and *tentative* $= \{v \mid v \text{ is covered}\}$.

Proposition 4. *Procedure* EXPLORE *is partially correct: if* EXPLORE$(\mathcal{A}, \ell_e)$ *terminates, then the result is* SAFE *iff* ℓ_e *is unreachable for* $\mathcal{A}$.

Proof (sketch). The main loop in line 10 maintains the following invariants:

1. $V =$ *passed* $\cup$ *waiting* $\cup$ *tentative* $\cup$ *infeasible*,
2. *passed* is a set of non-excluded, expanded, ℓ_e-safe, well-labeled nodes,
3. *waiting* is a set of non-excluded leaves that satisfy all conditions of well-labeledness, except maybe *simulation*,
4. *tentative* is a set of feasible, covered, ℓ_e-safe, well-labeled leaves, and
5. *infeasible* is a set of infeasible, ℓ_e-safe, well-labeled leaves.

Algorithm 1 Lazy reachability algorithm for timed automata

```
1: ensure $\rho = \textsc{safe}$ iff $\ell_e$ is unreachable for $\mathcal{A}$
2: function $\textsc{explore}(\mathcal{A}, \ell_e)$ returns $\rho \in \{\textsc{safe}, \textsc{unsafe}\}$
3:     let $v_0$ be a node such that $\ell_{v_0} = \ell_0$, $Z_{v_0} = Z_0$ and $W_{v_0} = \top$
4:     $V \leftarrow \{v_0\}$
5:     $E \leftarrow \emptyset$
6:     $\rhd \leftarrow \emptyset$
7:     let $G$ be an ASG for $\mathcal{A}$ over $V$, $E$ and $\rhd$
8:     $\mathit{passed} \leftarrow \emptyset$
9:     $\mathit{waiting} \leftarrow \{v_0\}$
10:    while $v \in \mathit{waiting}$ for some $v$ do
11:        $\mathit{waiting} \leftarrow \mathit{waiting} \setminus \{v\}$
12:        if $Z_v \sqsubseteq \bot$ then
13:            $\textsc{refine}(v, \bot)$
14:        else if $\ell_v = \ell_e$ then
15:            return $\textsc{unsafe}$
16:        else if there exists $v' \in \mathit{passed}$ such that $\ell_{v'} = \ell_v$ and $Z_v \sqsubseteq W_{v'}$ then
17:            $\textsc{cover}(v, v')$
18:        else
19:            $\textsc{expand}(v)$
20:    return $\textsc{safe}$

21: require $Z_v \sqsubseteq W_{v'}$
22: procedure $\textsc{cover}(v, v')$
23:     $\textsc{refine}(v, W_{v'})$
24:     if $W_v \sqsubseteq W_{v'}$ then
25:         $\rhd \leftarrow \rhd \cup \{(v, v')\}$
26:     else
27:         $\mathit{waiting} \leftarrow \mathit{waiting} \cup \{v\}$

28: procedure $\textsc{expand}(v)$
29:     for all $t \in T$ such that $t = (\ell_v, g, R, \ell')$ do
30:         let $v'$ be a new node such that $\ell_{v'} = \ell'$, $Z_{v'} = \mathbf{post}_t(Z_v)$ and $W_{v'} = \top$
31:         let $(v, v')$ be a new edge such that $t_{v,v'} = t$
32:         $V \leftarrow V \cup \{v'\}$
33:         $E \leftarrow E \cup \{(v, v')\}$
34:         $\mathit{waiting} \leftarrow \mathit{waiting} \cup \{v'\}$
35:     $\mathit{passed} \leftarrow \mathit{passed} \cup \{v\}$

36: require $Z_v \sqsubseteq W$
37: ensure $W_v \sqsubseteq W$
38: procedure $\textsc{refine}(v, W)$
39:     for all $B \in \neg W$ do
40:         $\textsc{block}(v, B)$

41: require $Z_v \sqcap B \sqsubseteq \bot$
42: ensure $W_v \sqcap B \sqsubseteq \bot$
43: procedure $\textsc{block}(v, B)$
```

It is easy to verify that under the above assumptions, these sets form a partition of V. Partial correctness of the algorithm is then a direct consequence. Since at line 20 the set *waiting* is empty, so G is complete, well-labeled and ℓ_e-safe, and as a consequence of Lemma 2, the location ℓ_e is indeed unreachable for $\mathcal{A}$. Conversely, at line 15, a node is encountered that is non-excluded, well-labeled and not ℓ_e-safe, with all its ancestors well-labeled, thus by Lemma 3, there is a symbolic run of $\mathcal{A}$ to ℓ_e.

Building on the assumption that EXPAND, COVER and REFINE preserve the conditions of well-labeledness, showing that the loop invariant holds is straightforward. For EXPAND and COVER, this assumption can be easily proved. For REFINE, we need to prove that BLOCK preserves the conditions of well-labeledness. As calls to BLOCK might strengthen the labeling, care must be taken that the conditions (and in particular, *initiation(b)* and *consecution(b)*) are maintained In Section 4, this assumption is proved to hold. □

Termination, hence total correctness of the algorithm in this form can not be established, however, with the additional restriction in Remark 1, termination can be guaranteed. This is because refinement progress is ensured by the algorithm. After each call to COVER, either a node v gets covered, or a node $v' \in$ *passed* gets strengthened. As a node v' does not get strengthened beyond $Z_{v'}$, eventually, either all leaves become covered, an error node gets discovered, or a leaf gets expanded.

4 Abstraction refinement

To maintain well-labeledness, procedure REFINE relies on a procedure BLOCK that performes abstraction refinement by safely adjusting labels of nodes (see the reachability algorithm in Section 3.2). In this section, we propose two methods for abstraction refinement based on interpolation for zones.

4.1 Interpolation for zones

Let A and B be two canonical DBMs such that $A \sqcap B \sqsubseteq \bot$. An interpolant for the pair (A, B) is a canonical DBM I such that

- $A \sqsubseteq I$,
- $I \sqcap B \sqsubseteq \bot$, and
- clocks constrained in I are constrained in both A and B.

This definition of a DBM interpolant is analogous to the definition of an interpolant in the usual sense [14]. As DBMs encode formulas in $\mathcal{DL}(\mathbb{Q})$, a theory that admits interpolation [7], an interpolant always exists for a pair of inconsistent DBMs. Algorithm 2 is a direct adaptation of the graph-based algorithm of [7] for DBMs. For simplicity, we assume that A and B are defined over the same set of clocks with the same ordering, and are both canonical. Naturally, these restrictions can be lifted. For a more general description, see [17].

Algorithm 2 Interpolation for zones represented as canonical DBMs

```
 1: require $A \sqcap B \sqsubseteq \bot$
 2: ensure $A \sqsubseteq I$
 3: ensure $I \sqcap B \sqsubseteq \bot$
 4: function INTERPOLATE$(A, B)$ returns $I$
 5:     if $A \sqsubseteq \bot$ then
 6:         return $\bot$
 7:     else if $B \sqsubseteq \bot$ then
 8:         return $\top$
 9:     else
10:         let $D = \min(A, B)$
11:         let $C = \{(i_1, i_2), \dots, (i_{k-1}, i_k), (i_k, i_1)\}$ be a negative cycle in $D$
12:         let $C_A = \{(i, j) \in C \mid A_{ij} = D_{ij}\}$
13:         let $I_{ij} = \begin{cases} (0, \leq) & \text{if } i = j \\ A_{ij} & \text{if } (i, j) \in C_A \\ \infty & \text{otherwise} \end{cases}$
14:         let $I = [I_{ij}]_{ij}$
15:         return $I$
```

After checking the trivial cases, the algorithm searches for a negative cycle in $\min(A, B)$ to witness its inconsistency. This can be done e.g. by running a variant of the Floyd-Warshall algorithm. As $A \sqcap B$ is inconsistent, such a cycle C exists by Proposition 2. Then the set C_A of edges that come from A is constructed. We can assume that no two such edges are subsequent, as A is canonical. Thus the DBM I induced by the corresponding constraints of A is clearly canonical. Moreover, it is easy to verify that I is indeed an interpolant.

4.2 Interpolation strategies for abstraction refinement

We propose two methods for abstraction refinement based on zone interpolation. Both methods are based on pre- and post-image computation, and can be considered as a generalization of zone interpolation to sequences of transitions of a timed automaton.

Conceptually, both methods for BLOCK work as follows. Given a node v and a zone B for which $Z_v \sqcap B \sqsubseteq \bot$ holds, a zone inconsistent with B is computed in form of an interpolant that is used to strengthen the current labeling. Meanwhile, conditions for well-labeledness are maintained. The condition of *coverage* is maintained by procedure STRENGTHEN that removes covering edges that would violate the condition after strengthening. However, the two methods differ in the strategy to ensure conditions *initiation(b)* and *consecution(b)*.

Algorithm 3 depicts the pseudocode for the two methods. We will refer to procedure BLOCK$_{\text{SEQ}}$ as the sequence (SEQ) strategy, and to procedure BLOCK$_{\text{BIN}}$ as the binary (BIN) strategy. The main difference is that BIN only applies backward propagation for refinement, whereas SEQ also uses forward propagation. We

Algorithm 3 Interpolation strategies for abstraction refinement

```
 1: require $Z_v \sqsubseteq I$
 2: ensure $W_v \sqsubseteq I$
 3: procedure STRENGTHEN$(v, I)$
 4:     for all $u$ such that $u \rhd v$ and $W_u \not\sqsubseteq I$ do
 5:         $\rhd \leftarrow \rhd \setminus (u, v)$
 6:         $\mathit{waiting} \leftarrow \mathit{waiting} \cup \{u\}$
 7:     $W_v \leftarrow W_v \sqcap I$

 8: require $Z_v \sqcap B \sqsubseteq \bot$
 9: ensure $W_v \sqsubseteq I$
10: ensure $W_v \sqcap B \sqsubseteq \bot$
11: function BLOCK$_{\text{SEQ}}(v, B)$ returns $I$
12:     if $W_v \sqcap B \sqsubseteq \bot$ then
13:         return $W_v$
14:     else
15:         if $(u, v) \in E$ for some $u$ then
16:             let $t = t_{u,v}$
17:             let $B' = \mathbf{pre}_t(B)$
18:             let $A' =$ BLOCK$_{\text{SEQ}}(u, B')$
19:             let $A = \mathbf{post}_t(A')$
20:         else
21:             let $A = Z_v$
22:         let $I =$ INTERPOLATE$(A, B)$
23:         STRENGTHEN$(v, I)$
24:         return $I$

25: require $Z_v \sqcap B \sqsubseteq \bot$
26: ensure $W_v \sqcap B \sqsubseteq \bot$
27: procedure BLOCK$_{\text{BIN}}(v, B)$
28:     if $W_v \sqcap B \sqsubseteq \bot$ then
29:         return
30:     else
31:         let $A = Z_v$
32:         let $I =$ INTERPOLATE$(A, B)$
33:         if $(u, v) \in E$ for some $u$ then
34:             let $t = t_{u,v}$
35:             for all $B'' \in \neg I$ do
36:                 let $B' = \mathbf{pre}_t(B'')$
37:                 BLOCK$_{\text{BIN}}(u, B')$
38:         STRENGTHEN$(v, I)$
```

show that both procedures are correct w. r. t. the annotations in Algorithm 3 and maintain well-labeledness.

Proposition 5. *Both variants of* BLOCK *are totally correct: if $Z_v \sqcap B \sqsubseteq \bot$, then* BLOCK$(v, B)$ *terminates and ensures $W_v \sqcap B \sqsubseteq \bot$. Moreover, they maintain well-labeledness.*

Proof. Termination of both methods is trivial, so we focus on partial correctness and the preservation of well-labeledness.

For BLOCK$_{\text{BIN}}$, if $W_v \sqcap B \sqsubseteq \bot$, then no strengthening is needed. If v is a root, it is easy to see that *initiation(b)* is maintained, and the postcondition trivially holds. Otherwise, after the loop, $W_u \sqcap \mathbf{pre}_t(B'') \sqsubseteq \bot$ for all $B'' \in \neg I$ by contract. Thus $\mathbf{post}_t(W_u) \sqcap B'' \sqsubseteq \bot$ for all $B'' \in \neg I$ by Lemma 1. Hence $\mathbf{post}_t(W_u) \sqsubseteq I$, so *consecution(b)* is maintained for v after strengthening. Moreover, $I \sqcap B \sqsubseteq \bot$, thus B is successfully blocked.

For BLOCK$_{\text{SEQ}}$, if $W_v \sqcap B \sqsubseteq \bot$, then no strengthening is needed. If v is a root, it is easy to see that *initiation(b)* is maintained, and the postconditions trivially hold. Otherwise A' is such that $A' \sqcap \mathbf{pre}_t(B) \sqsubseteq \bot$ by contract, thus $A \sqcap B \sqsubseteq \bot$ by Lemma 1. Thus the interpolant I can be computed, and $\mathbf{post}_t(A') \sqsubseteq I$.

Moreover, $W_u \sqsubseteq A'$ by contract, thus $\mathbf{post}_t(W_u) \sqsubseteq \mathbf{post}_t(A')$ by monotony of **post**. Hence $\mathbf{post}_t(W_u) \sqsubseteq I$, so *consecution(b)* is maintained for v after strengthening. Moreover, $I \sqcap B \sqsubseteq \bot$, thus B is successfully blocked. $\square$

5 Evaluation

We implemented a prototype version of Algorithm 1 in Java as an instantiation of the open source model checking framework Theta[1]. The only optimization we applied in the implementation compared to the presented algorithm is how coverage is handled: in the implementation, refine is only called if no covering node is present. Moreover, we implemented the two interpolation-based refinement strategies described in Algorithm 3.

For comparison, we also implemented a version of the lazy refinement algorithm of [11] based on LU-bounds ($\mathbf{a}_{\preccurlyeq LU}, \mathsf{disabled}$). The main difference in our implementation compared to [11] is that bounds are propagated from all guards on an infeasible path, and not just from ones that contribute to the infeasibility. Because of this, refinement in the resulting algorithm is extremely cheap, but as the comparison of our data with that of [11] suggests, for the examined models, the algorithm is still at least as space- and time-efficient as the original one. In some aspects, this refinement strategy is the opposite of interpolation based refinement: it provides a very cheap, non-convex, specialized refinement algorithm, as opposed to a relatively costly, convex, more general strategy. Apart from the abstraction and refinement strategy used ($\mathbf{a}_{\preccurlyeq LU}$, bin or seq), the three implementations of Algorithm 1 are identical.

Table 1 reports the results of our experiments. It contains the execution time (in seconds) and the final sizes of sets V and *passed*. The execution time is the average of 10 runs, obtained from 12 runs by removing the slowest and the fastest one. The input models are based on the PAT benchmarks[2]. For each model, the more efficient of BFS and DFS was applied as search order, which is BFS for all models except FDDI. We performed the measurements on a machine running Windows 10 with a 2.6GHz dual core CPU and 8GB of RAM.

For CSMA, FDDI, Fischer and Lynch, the three algorithms generated and expanded the same number of nodes. For FDDI, Fischer and Lynch, all three algorithms are optimal in this sense: the number of expanded nodes equals the number of distinct discrete states (plus one for FDDI), that is, clock variables do not influence the size of the ASG.

With respect to execution time, Fischer and Lynch provide the worst cases for our algorithm. The reason for the higher execution time despite the same number of generated nodes is that for these two models, the more costly refinement was not counterweighed by the smaller number of refinements performed, as opposed to CSMA, where the interpolation-based algorithms performed (as our logs showed) significantly less refinement steps. For FDDI, the three algorithms performed the same small number of refinement steps each, which explains the

[1] `http://theta.inf.mit.bme.hu`
[2] `http://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html`

Table 1. Comparison of lazy reachability algorithms

Model	$\mathbf{a}_{\preccurlyeq LU}$			BIN			SEQ		
	time	*nodes*	*passed*	*time*	*nodes*	*passed*	*time*	*nodes*	*passed*
Critical 3	1.8	23428	4923	1.6	14377	3213	1.6	14075	3157
Critical 4	65.0	838213	130779	78.2	536733	83686	75.2	499245	78252
CSMA 9	6.6	99207	30476	7.3	99207	30476	7.9	99207	30476
CSMA 10	21.3	251749	78605	21.0	251749	78605	22.8	251749	78605
CSMA 11	61.4	625215	198670	58.9	625215	198670	63.8	625215	198670
CSMA 12	167.2	1525525	493583	168.7	1525525	493583	179.1	1525525	493583
FDDI 50	1.4	504	402	2.0	504	402	2.0	504	402
FDDI 70	2.9	704	562	3.5	704	562	3.7	704	562
FDDI 90	5.9	904	722	6.8	904	722	7.1	904	722
FDDI 120	12.9	1204	962	15.0	1204	962	15.4	1204	962
Fischer 7	1.9	31060	7737	2.8	31060	7737	2.8	31060	7737
Fischer 8	5.1	111825	25080	7.7	111825	25080	8.7	111825	25080
Fischer 9	21.3	395956	81035	29.0	395956	81035	32.4	395956	81035
Fischer 10	94.4	1382921	260998	133.2	1382921	260998	149.7	1382921	260998
Lynch 7	2.6	51570	9977	3.6	51570	9977	4.0	51570	9977
Lynch 8	7.7	179273	30200	12.2	179273	30200	13.9	179273	30200
Lynch 9	32.8	620236	92555	45.2	620236	92555	54.2	620236	92555

slight relative overhead of the interpolation-based algorithms. However, the three algorithms scale in the same way.

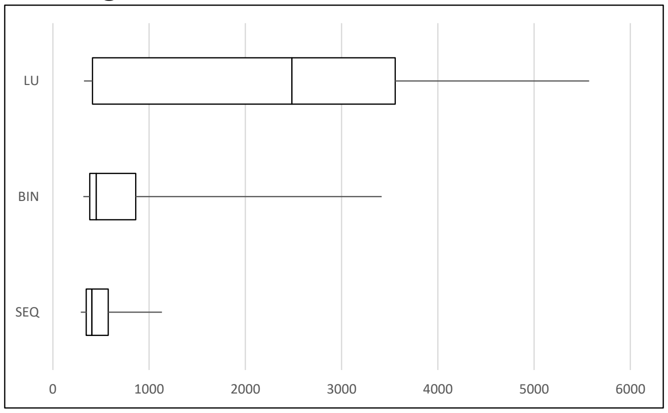
A favorable case for our algorithm with respect to ASG size is provided by the model Critical. For this model, the interpolation-based algorithms were able to generate a 40% smaller ASG as $\mathbf{a}_{\preccurlyeq LU}$, with a 15-20% relative overhead in execution time. Among the two interpolation strategies, SEQ was somewhat more efficient in both aspects.

We also evaluated the three methods under random search order. We used FDDI as an input model, as this model is known to be sensitive to search order: with the right abstraction and search order, it scales linearly in the number of processes (as in Table 1), otherwise, it scales exponentially. The results of our experiment are shown on the boxplot in Figure 1, which depicts the ASG sizes for 50 random runs of each algorithm. As the boxplot suggests, the interpolation-based refinement methods, and SEQ in particular, are less sensitive to search order with respect to the size of the generated tree, and are better at recovering from bad decisions during search.

6 Conclusions

In this paper, we proposed a lazy reachability checking algorithm for timed automata based on interpolation for zones. Moreover, we proposed two refinement strategies, both a combination of forward search, backward search and interpolation. We demonstrated with experiments that - even without the use of extrap-

Fig. 1. ASG size for random search of FDDI 10



olation - the method is competitive with sophisticated non-convex abstractions in both execution time and memory consumption.

Future work. As the method we proposed computes abstractions in terms of zones, it is straightforward to combine it with existing zone-based abstractions for timed automata. In particular, we believe that a combination with $\mathbf{a}_{\preccurlyeq LU}$, disabled would potentially yield a more efficient method with no considerable overhead, as backward propagation of LU-bounds is much cheaper than the propagation of interpolants. In this setting, interpolation can be considered as a further reduction on top of $\mathbf{a}_{\preccurlyeq LU}$ abstraction.

An interesting application of our approach would be to apply it to more expressive variants of timed automata, e.g. to automata with diagonal constraints in guards [6], or to updatable timed automata [5] with updates of the form $x_i := c$, $x_i := x_i + c$ (shift), $x_i := x_j$ (copy) or, more generally, even $x_i := x_j + c$. As all these operations yield zones both for forward and backward computation, with a generalization of **pre** and **post**, the approach becomes directly applicable. Naturally, due to general undecidability and the lack of a suitable extrapolation operator, termination can not be guaranteed in some of these cases [5].

We note that by switching the role of **pre** and **post** in the algorithm, a variant can be obtained that performs backward exploration in a lazy manner. Such an algorithm might result in an interesting method for simple timed automata with a restricted use of integer operations.

There are also many possibilities for fine-tuning the proposed algorithm. For example, the algorithm as described applies an aggressive covering strategy, as it tries all possible nodes for coverage before expanding a node. The investigation of more sophisticated covering strategies (e.g. forced covering as in [15]) might yield better scaling with respect to execution time. Moreover, our current implementation is based on DBMs. The adaptation of the method to e.g. minimal constraint systems is straightforward, and is possibly more efficient.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2619, pp. 254–270. Springer (2003)
3. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 2988, pp. 312–326. Springer (2004)
4. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer (2004)
5. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods in System Design 24(3), 281–320 (2004)
6. Bouyer, P., Laroussinie, F., Reynier, P.A.: Diagonal constraints in timed automata: Forward analysis of timed systems. In: Formal Modeling and Analysis of Timed Systems. LNCS, vol. 3829, pp. 112–126. Springer (2005)
7. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theories. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 4963, pp. 397–412. Springer (2008)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM 50(5), 752–794 (2003)
9. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 1384, pp. 313–329. Springer (1998)
10. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages. pp. 58–70. ACM (2002)
11. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. In: Computer Aided Verification. LNCS, vol. 8044, pp. 990–1005. Springer (2013)
12. Herbreteau, F., Kini, D., Srivathsan, B., Walukiewicz, I.: Using non-convex approximations for efficient analysis of timed automata. In: Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 13, pp. 78–89 (2011)
13. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. In: Logic in Computer Science. pp. 375–384. LICS, IEEE (2012)
14. McMillan, K.L.: Interpolation and sat-based model checking. In: Computer Aided Verification. LNCS, vol. 2725 LNCS, pp. 1–13. Springer (2003)
15. McMillan, K.L.: Lazy abstraction with interpolants. In: Computer Aided Verification. LNCS, vol. 4144 LNCS, pp. 123–136. Springer (2006)
16. McMillan, K.L.: Lazy annotation for program testing and verification. In: Computer Aided Verification. LNCS, vol. 6174, pp. 104–118. Springer (2010)
17. Tóth, T., Majzik, I.: Timed automata verification using interpolants. In: Proceedings of the 24th PhD Mini-Symposium. pp. 82–85. BUTE DMIS (2017), `http://oszkdk.oszk.hu/DRJ/19248`
18. Wang, W., Jiao, L.: Difference bound constraint abstraction for timed automata reachability checking. In: Formal Techniques for Distributed Objects, Components, and Systems. LNCS, vol. 9039, pp. 146–160. Springer (2015)