Lazy abstractions for timed automata

F. Herbreteau[1], B. Srivathsan[2], and I. 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 is updating 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 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 the execution of a given automaton. The standard solution to this problem involves computing, so called, zone graph of the automaton, and the use of 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 [AD94]. Behrmann et. al. [BBLP06] have proposed abstractions based on so called LU-bounds, that are functions giving for every clock a maximal constant in a lower, respectively upper bound, constraint in the automaton. In a recent paper [HSW12] we have shown how to efficiently use $\mathfrak{a}_{\preccurlyeq LU}$ abstraction from [BBLP06]. 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 op. cit. 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.

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 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. In the 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 static analysis algorithm that is the state-of-the-art algorithm implemented in UPPAAL, and with an algorithm we have proposed in [HKSW11]. The later improves on the static analysis algorithm by considering only the reachable part of the zone graph. For an example borrowed from [LNZ05] 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, that has been tested on just about every algorithm for the reachability problem, our algorithm shows rather surprising fact that the time 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 the spirit to [HJMS02], 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 [DT98]. The notion of LU-bounds and inference of these bounds by static analysis of an automaton have been proposed in [BBFL03,BBLP06]. The $\mathfrak{a}_{\preccurlyeq_{LU}}$ approximation has been introduced in [BBLP06]. An approximation method based on LU-bounds, called $Extra^{+}_{LU}$, is used in the current implementation of UPPAAL [BDL+06]. In [HSW12] we have shown how to effi-

ciently use $\mathfrak{a}_{\preccurlyeq_{LU}}$ approximation. We have also proposed an LU-propagation algorithm [HKSW11] that can be seen as applying the static analysis from [BBFL03] 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 analysis of timed-automata have been considered almost immediately after introduction of the concept of timed automata, as for example in [WT94,DWT95] or [Sor04]. In particular, the later citation proposes 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 state-of-the-art method used in UPPAAL. Another slightly related paper is [BLR05] where CEGAR approach is used to handle diagonal constraints.

Let us mention that abstractions are not needed in backward exploration of timed systems. Nevertheless, any feasible backward analysis approach needs to simplify constraints. For example [MPS11] does not use approximations and relies on an SMT solver instead. This approach, or the approach of RED [Wan04], 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 algorithm on variations of an example borrowed from [LNZ05]. The experiments section compares our prototype tool with UPPAAL, and our algorithm from [HKSW11]. Conclusions section gives some justification for our choice of concentrating on LU-bounds.

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 $v : X \to \mathbb{R}_{\geq 0}$. We denote $\mathbb{R}^X_{\geq 0}$ the set of clock valuations over X, and $\mathbf{0}$ the valuation that associates 0 to every clock in X. We write $v \vDash \phi$ when v satisfies $\phi \in \Phi(X)$, i.e. when every constraint in ϕ holds after replacing every x

by $v(x)$. 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 $[R]v$ 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 $(q, v) \in Q \times \mathbb{R}^X_{\geq 0}$ 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: $(q, v) \to^{t} (q, v')$ for some transition $t = (q, g, R, q') \in T$ such that $v \vDash g$ and $v' = [R]v$.

A *run* of $\mathcal{A}$ is a finite sequence of transitions starting from the initial configuration $(q_0, \mathbf{0})$. Without loss of generality, we can assume that the first transition is a delay transition and that delay and action transitions alternate. We write $(q, v) \xrightarrow{\delta, t} (q', v')$ if there is a delay transition $(q, v) \to^{\delta} (q, v+\delta)$ followed by an action transition $(q, v+\delta) \to^{t} (q', v')$. So a run of $\mathcal{A}$ can be written as:

$$(q_0, v_0) \xrightarrow{\delta_0, t_0} (q_1, v_1) \xrightarrow{\delta_1, t_1} (q_2, v_2) \dots (q_n, v_n)$$

where (q_0, v_0) represents 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 [AD94,CY92]. 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.

2.2 Zones and symbolic runs

Here we introduce zones that are sets of valuations defined by simple linear constraints. We also define symbolic transition relation working on sets of valuations. These definitions will allow us to concentrate on symbolic runs instead of concrete runs as in the previous section.

We first define a transition relation $\Rightarrow$ over nodes of the form (q, W) where W is 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 have a transition $\Rightarrow^t$ defined as follows:*

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

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

The transition relation defined above considers each valuation $v \in W$ that can take the transition t, obtains the valuation after the transition and then collects the time-successors from this obtained valuation. Therefore the symbolic transition $\Rightarrow$ always yields sets closed under time-successors. The initial configuration of the automaton is $(q_0, \mathbf{0})$. Starting from the initial valuation $\mathbf{0}$ the set of valuations reachable by a time elapse at the initial state are given by $\{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$. Call this W_0. From (q_0, W_0) as the initial node, computing the symbolic transition relation $\Rightarrow$ leads to different nodes (q, W) wherein the sets W are closed under time-successors.

It has been noticed in [BY04a] that the sets W obtained in the nodes (q, W) can be described by some simple constraints involving only the difference between clocks. This has motivated the definition of *zones*, which are sets of valuations defined by difference constraints.

Definition 3 (Zones [BY04a]). *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* $x, y \in X$*,* $\sim \in \{\leq, <, =, >, \geq\}$*, and* $c \in \mathbb{Z}$*.*

For example $(x > 4 \wedge y - x \leq 1)$ is a zone. It can be shown that starting from a node (q, W) with W being a zone, the transition $(q, W) \Rightarrow (q', W')$ leads to a node in which W' is again a zone [BY04a]. Observe that the initial set of valuations $Z_0 = \{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$ is indeed a zone: it is given by the constraints $\bigwedge_{x,y \in X} \ (x \geq 0 \ \wedge \ x - y = 0)$

These observations lead to a notion of *symbolic run* that is a sequence of symbolic transitions

$$(q_0, Z_0) \Rightarrow (q_1, Z_1) \Rightarrow \dots$$

Proposition 1. *Fix a timed automaton. The automaton has an accepting run if and only if there it has a symbolic run reaching an accepting state and non-empty zone.*

This proposition does not yet give a complete solution to the reachability problem since there may be infinitely many reachable zones, so it is not immediate how to algorithmically check that a symbolic run does not exist. A standard solution to this problem of non-termination is to use abstractions that we introduce in the next subsection.

2.3 Bounds and abstractions

In the previous subsection, we have defined zones. We have used zones instead of valuations to solve the reachability problem. Since the number of reachable zones can be infinite, the next step is to group zones together into a finite

number of equivalence classes. An *abstraction operator* is a convenient way to express a grouping of valuations, and in consequence grouping of zones. Instead of discussing abstractions in full generality, we will immediately proceed to the most relevant case of abstractions based on time-abstract simulation [TAKB96].

For this subsection we fix a timed automaton $\mathcal{A}$.

Definition 4 (Time-abstract simulation). *A* (state based) time-abstract simulation *between configurations of $\mathcal{A}$ is a relation $(q,v) \preceq_{t.a.} (q',v')$ such that:*

- $q = q'$,
- *if* $(q,v) \to^{\delta} (q, v+\delta) \to^{t} (q_1, v_1)$*, then there exists a* $\delta' \in \mathbb{R}_{\geq 0}$ *such that* $(q,v') \to^{\delta'} (q, v'+\delta') \to^{t} (q_1, v_1')$ *satisfying* $(q_1,v_1) \preceq_{t.a.} (q_1,v_1')$ *for the same transition* t*.*

For two valuations v, v', we say that $v \preceq_{t.a.} v'$ if for every state q of the automaton, we have $(q,v) \preceq_{t.a.} (q',v')$. An abstraction $\mathfrak{a}_{\preceq_{t.a.}}$ based on a simulation $\preceq_{t.a.}$ can be defined as follows:

Definition 5 (Abstraction based on simulation). *Given a set W, we define $\mathfrak{a}_{\preceq_{t.a.}}(W) = \{v \mid \exists v' \in W.\ v \preceq_{t.a.} v'\}$. The abstract transition relation is $(q,W) \Rightarrow_{\mathfrak{a}_{\preceq_{t.a.}}} (q', \mathfrak{a}_{\preceq_{t.a.}}(W'))$ where $W = \mathfrak{a}_{\preceq_{t.a.}}(W)$ and $(q,W) \Rightarrow (q',W')$ (cf. Definition 2).*

Let $\Rightarrow^*_{\mathfrak{a}_{\preceq_{t.a.}}}$ denote the reflexive and transitive closure of $\Rightarrow_{\mathfrak{a}_{\preceq_{t.a.}}}$. Similarly, let $\to^*$ denote the reflexive and transitive closure of the transition relation $\to$ of the automaton. It can be easily verified that the abstract transition relation satisfies the following two important properties (W_0 denotes $\{\mathbf{0}+\delta \mid \delta \in \mathbb{R}_{\geq 0}\}$)

Soundness: if $(q_0, W_0) \Rightarrow^*_{\mathfrak{a}_{\preceq_{t.a.}}} (q,W)$ then there is $v \in W$ such that $(q_0, \mathbf{0}) \to^* (q,v)$.

Completeness: if $(q_0, \mathbf{0}) \to^* (q,v)$ then there is W such that $v \in W$ and $(q_0, W_0) \Rightarrow^*_{\mathfrak{a}_{\preceq_{t.a.}}} (q,W)$.

These properties immediately imply that abstract transitions can be used to solve the reachability problem.

Proposition 2. *For every abstraction operator $\mathfrak{a}_{\preceq_{t.a.}}$ based on timed-abstract simulation. Automaton $\mathcal{A}$ has a run reaching a state q iff there is an abstract run*

$$(q^0, W_0) \Rightarrow_{\mathfrak{a}_{\preceq_{t.a.}}} (q_1, W_1) \Rightarrow_{\mathfrak{a}_{\preceq_{t.a.}}} \dots \Rightarrow_{\mathfrak{a}_{\preceq_{t.a.}}} (q, W)$$

for some $W \neq \emptyset$.

Remark 1. If $\mathfrak{a}$ and $\mathfrak{b}$ are two abstractions such that for every set of valuations W we have $\mathfrak{a}(W) \subseteq \mathfrak{b}(W)$ then we prefer to use $\mathfrak{b}$ since every abstract run with respect to $\mathfrak{a}$ is also a run with respect to $\mathfrak{b}$. In consequence, it is easier to find an abstract run for $\mathfrak{b}$ abstraction.

Therefore, the aim is to come up with a finite abstraction as coarse as possible, that still maintains the soundness property.

For a given automaton it can be computed if two configurations are in a simulation relation. It should be noted though that computing the coarsest simulation relation is EXPTIME-hard [LS00]. Since the reachability problem can be solved in PSPACE, this suggests that it may not be reasonable to try to solve it using the abstraction based on the coarsest simulation. We can get simulation relations that are computationally easier if we consider only a part of the structure of the automaton. The common way is to look at constants appearing in the guards of the automaton and consider them as parameters for abstraction.

2.4 LU-bounds and LU-abstractions

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

Definition 6 (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, 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, maximum over guards of the form $x < c$ or $x \leq c$.*

The paper introducing LU-bounds [BBLP06] 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 [BBLP06]. We use a different but equivalent formulation.

Definition 7 (LU-preorder [BBLP06]). *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 $v(x) > U_x$.*

It has been shown in [BBLP06] that $\preccurlyeq_{LU}$ is a time-abstract simulation relation. The $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction is based on this LU-preorder $\preccurlyeq_{LU}$.

Definition 8 ($\mathfrak{a}_{\preccurlyeq_{LU}}$-abstraction [BBLP06]). *Given L and U bound functions, for a set of valuations W we define:*

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

Figure 1 gives an example of a zone Z and its abstraction $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$. It can be seen that $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ is not a convex set.

An efficient algorithm to use the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction for reachability was proposed in [HSW12]. 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

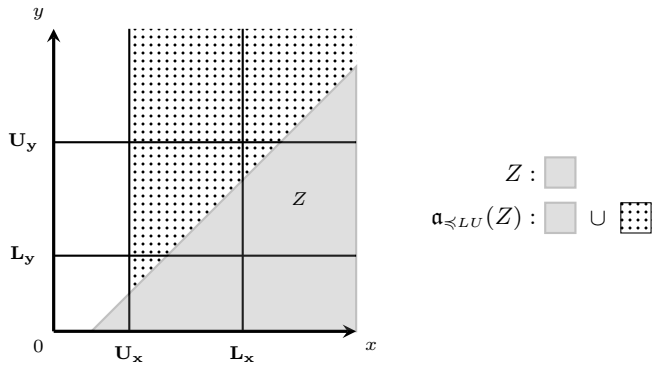

Fig. 1. Zone Z is given by the grey area. Abstraction $\mathfrak{a}_{\preccurlyeq LU}(Z)$ is given by the grey area along with the dotted area

$\mathfrak{a}_{\preccurlyeq LU}$ 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. Recall Remark 1 which states the importance of having coarser abstractions.

It has been proposed in [BBFL03] 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:

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

Similar inequalities are written for U, now considering $x \preccurlyeq c$. It has been shown in [BBFL03] 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 structure of 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 while searching for an abstract run. The immediate gain will be that bounds will depend not only on a state but also on a set of valuations. The real freedom given by an adaptive simulation graph and Theorem 1 presented below is that they will allow to ignore some guards of transitions when calculating the LU bounds. As we will see in experimental section, this can result in very big performance gains.

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

Definition 9 (Adaptive simulation graph (ASG)). *Fix an automaton $\mathcal{A}$. An ASG graph 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 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 *For every edge $(q, Z, LU) \Rightarrow_t (q', Z', L'U')$ the ASG we have:*

$$\mathrm{Post}_t(\mathfrak{a}_{\preccurlyeq_{LU}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq_{L'U'}}(Z').$$

I3 *For every tentative node (q, Z_1, L_1U_1) and the corresponding covering node (q, Z_2, L_2U_2), we have:*

$$L_2U_2 \leq L_1U_1.$$

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 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 $W = \mathfrak{a}_{\preccurlyeq_{LU}}(W)$, we do not want to store W directly, as we have no efficient way of representing and manipulating such 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 $Z' \subseteq Z$ [HSW12]. This approach has another big advantage: when we change 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 example, it is sufficient to take static LU-bounds as described in (1). It means that we can take ASG whose nodes are $(q, Z, L(q)U(q))$ with bound functions given by static analysis. 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 later section will be to construct as small ASG 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.*

Recall from Proposition 2 that there is an accepting run of $\mathcal{A}$ iff there is a sequence of symbolic transitions

$$(q_0, Z_0) \Rightarrow (q_1, Z_1) \Rightarrow \ldots \Rightarrow (q, Z) \tag{2}$$

with $q \in Acc$ and $Z \neq \emptyset$.

For the right-to-left direction of the theorem we take a path in G leading from (q_0, Z_0, L_0U_0) to (q, Z, LU). By definition, removing the third component gives us a path as in (2).

The opposite direction is proved with the help of the following lemma.

Lemma 1. *Let (q, Z) be as in (2). There exists a non tentative node (q, Z_1, L_1U_1) in G such that $Z \subseteq \mathfrak{a}_{\preccurlyeq L_1U_1}(Z_1)$.*

Proof. The lemma is vacuously true for (q_0, Z_0). Assume that the hypothesis is true for a path as in (2). We prove that the lemma is true for every symbolic successor of (q, Z).

Let $(q, Z) \Rightarrow^t (q', Z')$ be a symbolic transition of $\mathcal{A}$. The transition $\Rightarrow^t$ should be enabled from (q, Z_1). This is because if it was disabled, by Invariant 1, we would have that it is disabled from $\mathfrak{a}_{\preccurlyeq L_1U_1}(Z_1)$ and from the hypothesis, it should be disabled from (q, Z) too leading to a contradiction.

So we have a transition $(q, Z_1, L_1U_1) \Rightarrow^t (q', Z_1', L_1'U_1')$ in G. From Invariant 2, we have $\mathrm{Post}(\mathfrak{a}_{\preccurlyeq L_1U_1}(Z_1)) \subseteq \mathfrak{a}_{\preccurlyeq L_1'U_1'}(Z_1')$. This leads to the following sequence of implications.

$$\begin{array}{rrcll} & Z & \subseteq & \mathfrak{a}_{\preccurlyeq L_1U_1}(Z_1) & \text{induction hypothesis} \\ \Rightarrow & \mathrm{Post}(Z) & \subseteq & \mathrm{Post}(\mathfrak{a}_{\preccurlyeq L_1U_1}(Z_1)) & \\ \Rightarrow & \mathrm{Post}(Z) & \subseteq & \mathfrak{a}_{\preccurlyeq L_1'U_1'}(Z_1') & \text{by Invariant 2} \\ \Rightarrow & Z' & \subseteq & \mathfrak{a}_{\preccurlyeq L_1'U_1'}(Z_1') & \end{array}$$

If $(q', Z_1', L_1'U_1')$ is a non-tentative node, then we are done. Suppose it is a tentative node, then we know that there exists a non-tentative node $(q', Z_2', L_2'U_2')$ such that $Z_1' \subseteq \mathfrak{a}_{\preccurlyeq L_2'U_2'}(Z_2')$. From Invariant 3, we also know that $L_2'U_2' \leq L_1'U_1'$. This shows that $Z' \subseteq \mathfrak{a}_{\preccurlyeq L_2'U_2'}(Z_2')$.

Hence the node corresponding to (q', Z') is $(q', Z_2', L_2'U_2')$. □

4 Algorithm

Our aim is to construct a small adaptive simulation graph for a given timed automaton. For this the algorithm will try to keep LU bounds as small as possible but still satisfy the invariants I1, I2, I3. The bounds are calculated dynamically while constructing an adaptive simulation graph. For example, the invariant I1 requires LU in a node to be sufficiently big so that the transition remains disabled. Invariant I2 tells that LU bound in a node should depend on LU bounds in the successors of the node.

Proviso: For simplicity of the algorithm presented in this section we assume a special form of transitions of timed automata. A transition can have either only upper bound guards, or only lower bound guards 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.

Lemma 2. *Suppose W_1 is a time elapsed set of valuations. If*

$$(q_1, W_1) \Rightarrow_{g;R} (q_2, W_2) \quad \textit{and} \quad (q_1, W_1) \Rightarrow_{g_L} (q_1', W_1') \Rightarrow_{g_U;R} (q_2, W_2')$$

then $W_2 = W_2'$.

Proof. We consider only, more complicated, inclusion $W_2' \subseteq W_2$. Take $v_2' \in W_2'$. By definition we know that there is $v_1 \in W_1$ such that

$$(q_1, v_1) \xrightarrow{g_L} (q_1', v_1 + \delta_1) \xrightarrow{g_U;R} (q_2, (v_1 + \delta_1)[R] + \delta_2)$$

and $v_2 = (v_1 + \delta_1)[R] + \delta_2$. We get then

$$(q_1, v_1 + \delta_1) \xrightarrow{g_L} (q_1', v_1 + \delta_1) \xrightarrow{g_U;R} (q_2, (v_1 + \delta_1)[R] + \delta_2)$$

So $(q_1, v_1 + \delta_1) \xrightarrow{g;R} (q_2, (v_1 + \delta_1)[R] + \delta_2)$. As W_1 is time elapsed, $v_1 + \delta_1 \in W_1$. This shows $v_2 \in W_2$, by definition of W_2. □

So in order to satisfy our proviso we may need to double the number of states of an automaton.

Algorithm 1.1 presented below, computes a tree whose nodes v have 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 $\mathcal{A}$ from $(v.q, v.Z)$ resulting in a non-empty zone. 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 needed to be explored.

Our algorithm starts from the root node v_{root} labeled with q_0 and Z_0: the initial state of $\mathcal{A}$, and the initial zone. We do not set the LU bounds for v_{root}

Algorithm 1.1. Reachability algorithm with on-the-fly bound computation and $\mathfrak{a}_{\preccurlyeq LU}$ abstraction.

```
1  function main():
2      let v_root be the root node with v_root.q = q_0 and v_root.Z = Z_0
3      add v_root to the stack
4      while (stack ≠ ∅) do
5          remove v from the stack
6          explore(v)
7          resolve()
8      return "empty"
9
10 procedure explore(v):
11     if (v.q is accepting)
12         exit "not empty"
13     if (∃ v'' nontentative s.t. v.q = v''.q and v.Z ⊆ a_{≼v''.LU}(v''.Z))
14         mark v tentative wrt v''
15         v.LU := v''.LU
16         (X_L, X_U):= active clocks in v.LU
17         propagate(v, X_L, X_U)
18     else
19         v.LU := disabled(v.q, v.Z)
20         (X_L, X_U):= active clocks in v.LU
21         propagate(v, X_L, X_U)
22         for each (q', Z') s.t. (v.q, v.Z) ⇒ (q', Z') and Z' ≠ ∅ do
23           create v' the successor of v with v'.q = q' and v'.Z = Z'
24           explore(v')
25
26 function disabled(q, Z)
27     examine transitions from q that are disabled from Z and
28     choose LU so that invariant I1 is satisfied
29     return(LU);
30
31 procedure resolve():
32     for each v tentative w.r.t. v' do
33         if v.Z ⊈ a_{≼v'.LU}(v'.Z)
34             mark v nontentative
35             set v.L and v.U to −∞ // clear the bounds in v
36             add v to stack
37
38 procedure propagate(v', X'_L, X'_U):
39     v=parent(v');
40     LU := newbounds(v, v', X'_L, X'_U)
41     if (LU ≠ v.LU)
42         for each v_t tentative wrt v do
43             (X^t_L, X^t_U) clocks modified in LU wrt v_t.LU.
44             v_t.LU := LU;
45             propagate(v_t, X^t_L, X^t_U)
46         if (v ≠ v_root) then
47             (X_L, X_U) clocks modified in LU wrt v.LU
48             propagate(v, X_L, X_U)
49
50 function newbounds(v, v', X'_L, X'_U)
51  given a transition v → v', find new LU bounds for v knowing
52  that LU bounds for v' have changed, and
53  X'_L are the clocks whose L bound has changed,
54  X'_U are the clocks whose U bound has changed
```

as this will be done by `explore` procedure. The main loop repeatedly alternates an *exploration* and a *resolution* phases until there are no nodes to be explored. The exploration phase constructs a part of ASG from a given node stopping at nodes that it considers tentative. During exploration LU bounds of some nodes may be changed in order to preserve invariants I2 and I3. The resolution phase examines tentative nodes and adds them to the stack for exploration if condition G3 of the definition of ASG is no longer satisfied.

At the call of the procedure `explore`(v), node v is supposed to have its state $v.q$ and zone $v.Z$ set but the value of $v.LU$ is irrelevant. The zone $v.Z$ is supposed to be not empty. We assume that the constructed tree satisfies the invariants I1, I2, I3, but for the node v and the nodes on the stack. The goal of the `explore` procedure is to restore the invariant for v and start exploration of successors of v if needed.

First, the procedure checks if $v.q$ is an accepting state. If so then we know that this state is reachable since we assume that $v.Z$ is not empty. When $v.q$ is not accepting we consider two cases. If there exists a $non-tentative$ node v'' in the current tree such that $v.q = v''.q$ and $v.Z \subseteq \mathfrak{a}_{\preccurlyeq v''.LU}(v''.Z))$ then v is a tentative node. The LU-bounds from v'' are copied to v, and propagated so that invariant $I2$ is restored. This is the task of `propagate` procedure that we describe below. If v is not covered then it should be explored. First, we compute its LU bound based on transitions that are disabled from v. The task of function `disabled` is to calculate the LU bounds so that the invariant I1 holds. (The function is described in more detail in the next section.) Then we propagate these bounds in order to restore the invariant I2. Finally, we explore from every successor of v.

When LU bounds in a node v' are changed the invariant I2 should be restored. For this the bounds are propagated by invoking `propagate` procedure. For efficiency, the procedure is also given the set of clocks X'_L whose L bound has changed, and the set X'_U of clocks whose U bound has changed. The parent v of v' is taken and the transition from v to v' is examined. The function `newbounds` calculates new LU bounds for a node given the changes in its successor. This function is the core of our algorithm and is the subject of the next section. Here it is enough to assume that the new bounds are such that the invariant I2 is satisfied. If the bounds of v indeed change then they should be copied to all nodes tentative with respect to v. This is necessary to satisfy the invariant I3. Finally, the bounds are propagated to the predecessor of v to restore invariant I2.

The exploration phase terminates as in the `explore` procedure the bound functions in each node never decrease and are bounded. They are bounded because `newbounds` function never gives bounds bigger than those obtained by static analysis (cf. Equation (1))

After exploration phase LU bounds of tentative nodes may change. The 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, and put on the *stack* for further consideration in the function `main`.

The algorithm terminates when either it finds and 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 9).

From the above discussion it follows that the algorithm returns "*empty*" only when it constructs a complete ASG. The correctness of the algorithm then follows from Theorem 1.

Proposition 3. *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

The notion of adaptive simulation graph (Definition 9) 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 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 be not smaller than in the covering node. The algorithm from the last section implements a construction of ASG with updates of the bounds when the required by the invariant.

The three invariants sometimes allow for much smaller LU-bounds than that obtained by static analysis. A very simple example is when the algorithm does not encounter a node with a disabled edge. In this case all LU-bounds are simply $-\infty$, since no bound is increased due to I1, and such bounds are not changed by propagation. When LU bounds are $-\infty$, $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction of a zone results in the set of all valuations. So in this case ASG can be just a subgraph of the automaton. A more interesting examples of important gains are discussed in the next section.

In this section we describe two central functions of the proposed algorithm: `disabled` and `newbounds`. The pseudo-code is presented in Algorithm 1.2.

The `disabled` function is quite simple. Its task is to restore the invariant I1. For this it 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. 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 that is not satisfied in a zone. Now it suffices to observe that if a guard $x \leq d$ or $x < d$ is not satisfied in Z then it is not satisfied in $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ when $U(x) = d$. This follows directly from the definition of LU-simulation (Definition 7).

For the rest of this section we focus on the description of the function `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 function `newbounds` 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 9.

Recall that we have assumed that every transition has either only upper bound guards, or only lower bound guards and no resets (cf. page 10). This assumption will simplify the `newbounds` function. We will first consider the case of transitions with just an atomic guard or with just a reset. Next we will put what we have learned together to treat the general case.

5.1 Reset

Consider a transition $(q, Z) \Rightarrow_R (q', Z')$ for the set of clocks R being reset. So we have $Z' = \overrightarrow{Z[R := 0]}$, i.e., we reset the clocks in R and let the time elapse. Suppose that we have updated $L'U'$ and now we want our `newbounds` function to compute $L_{new}U_{new}$. We let $L_{new}U_{new}$ be the maximum of LU and $L'U'$ but for $L_{new}(x) = U_{new}(x) = -\infty$ for $x \in R$. We want to show that invariant I2 holds that is:

$$\mathfrak{a}_{\preccurlyeq_{L_{new}U_{new}}}(Z)[R := 0] \subseteq \mathfrak{a}_{\preccurlyeq_{L'U'}}(Z[R := 0]).$$

To prove this inclusion, take a valuation $v \in \mathfrak{a}_{\preccurlyeq_{L_{new}U_{new}}}(Z)$. By definition there is a valuation $v' \in Z$ with $v \preccurlyeq_{L_{new}U_{new}} v'$. We obtain that $v[R := 0] \preccurlyeq_{L'U'} v'[R := 0]$ using directly Definition 7. Indeed, for every clock in R, its values in the two valuations are the same. For other clocks the required implications hold since $v \preccurlyeq_{L_{new}U_{new}} v'$ and moreover the bounds $L_{new}U_{new}$ and $L'U'$ are the same for these clocks.

5.2 An abstract formula for atomic guard case

Consider a transition $(q, Z, LU) \Rightarrow^g (q', Z', L'U')$. Suppose that we have updated $L'U'$ and now we want our `newbounds` function to compute $L_{new}U_{new}$. In the standard constant propagation algorithm, 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 where there is only one constraint, there is only one constant associated to a single

clock by L_gU_g. Roughly, in order to maintain Invariant 2, it suffices to take

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U') & \text{if } LU \geq L_gU_g \text{ or} \\ & \text{if } [\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z') \text{ or} \\ & \text{if } Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z') \\ \max(LU, L'U', L_gU_g) & \text{otherwise} \end{cases} \tag{3}$$

To see why the above should maintain Invariant 2, look at the transition with the new bounds:

$$(q, Z, L_{new}U_{new}) \Rightarrow^g (q', Z', L'U')$$

Clearly from the above definition, $L_{new}U_{new} \geq L'U'$.

Additionally, if $L_{new}U_{new} \geq L_gU_g$, that is, if the constant in the guard is incorporated in $L_{new}U_{new}$, it is easy to show using definition of simulation that $\mathrm{Post}_g(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

We now need to show the same for the cases when $L_{new}U_{new}$ does not incorporate the constant in the guard. From the definition of the Pre, this happens only if either $g \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$ or if $Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. Let us look closely at what $\mathrm{Post}_g(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z))$ is.

$$\mathrm{Post}_g(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) = \overrightarrow{\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z) \cap [\![g]\!]}$$

If $[\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$, then $\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z) \cap [\![g]\!]$ would be included in $\mathfrak{a}_{\preccurlyeq L'U'}(Z')$. As $\mathfrak{a}_{\preccurlyeq L'U'}$ is closed under time-elapse, we will have $\mathrm{Post}_g(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. Similarly if $Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$, we will have $\mathfrak{a}_{\preccurlyeq L'U'}(Z) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$ and as $L_{new}U_{new} \geq L'U'$, we have $\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. It follows that $\mathrm{Post}_g(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

5.3 A concrete algorithm for atomic guard case

Since bound propagation is called very often in the main algorithm, we need an efficient test for the inclusions in Formula (3). The formula requires us to test inclusion w.r.t. $\mathfrak{a}_{\preccurlyeq LU}$ between Z and Z' each time we want to do the Pre. 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 $Z' = \overrightarrow{Z \wedge g}$. This makes the inclusion test 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 need to look closer how zones are represented. One standard way to represent zones is using difference bound matrices (DBMs) [Dil89]. We will consider 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 $x \xrightarrow{\prec c} y$

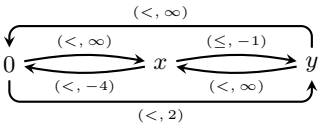

Fig. 2. Distance graph for the zone $(x - y \geq 1 \wedge y < 2 \wedge x > 4)$.

represents a constraint $y - x \prec c$: or in words, the distance from x to y is bounded by c. An example of a distance graph is depicted in Fig. 2.

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.

One can define an arithmetic and order over the weights $(\prec, c)$ in an expected manner [BY04b]. We recall only the definition of order that is most relevant for us here

$$\textit{Order } (\prec_1, c_1) < (\prec_2, c_2) \text{ if either } c_1 < c_2 \text{ or } (c_1 = c_2 \text{ and } \prec_1 =< \text{ and } \prec_2 =\leq).$$

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. For instance, the distance graph shown in Figure 2 is not in canonical form as the weight of the edge $x \to y$ is $(\leq, -1)$ whereas there is a path $x \to 0 \to y$ whose weight is $(<, -2)$. To convert it to canonical form, it is sufficient to change the weight of the edge $x \to y$ to $(<, -2)$.

For two distance graphs G_1, G_2 which are not necessarily in canonical form, we denote by $\min(G_1, G_2)$ the distance graph where each edge has the weight equal to the minimum of the corresponding weights in G_1 and G_2. Even though this graph may be not in canonical form, it should be clear that it represents intersection of the two arguments, that is, $[\![\min(G_1, G_2)]\!] = [\![G_1]\!] \cap [\![G_2]\!]$; in other words, the valuations satisfying the constraints given by $\min(G_1, G_2)$ are exactly those satisfying all the constraints from G_1 as well as G_2.

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 [HSW12] that permits to handle $Z \subseteq \mathfrak{a}_{\preccurlyeq L' U'}(Z')$ test efficiently.

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:*

$$Z_{x0} \geq (\leq, -U'_x) \textit{ and } Z'_{xy} < Z_{xy} \textit{ and } Z'_{xy} + (<, -L'_y) < Z_{x0} \tag{4}$$

We are ready to proceed with our analysis. We distinguish two cases depending on whether the guard g is of the form $w \succ d$ or $w \prec d$.

Lower bound guard: When we have a lower bound guard, the diagonals do not change during intersection and time-elapse. Hence we have $Z'_{xy} = Z_{xy}$ when both x and y are non-zero variables. This shows that (4) 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 as both $Z_{0y} = Z'_{0y} = (<, \infty)$. It remains us to consider the single case when y is 0. It boils down to checking if there exists a clock x such that:

$$Z_{x0} \geq (\leq, -U'_x) \text{ and } Z'_{x0} < Z_{x0} \tag{5}$$

In words the above test asks if there exists a clock x whose $x \xrightarrow{\lessdot -c} 0$ edge in Z has reduced in Z' and additionally the edge weight $(\lessdot, -c)$ in Z satisfies either $c < U'_x$ or $(\lessdot, c) = (\leq, U'_x)$. If such a clock exists, the definition of Pre in (3) suggests that we need to check if $[\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

Let us look at the distance graph of $[\![g]\!]$. It has an edge $w \xrightarrow{\lessdot -d} 0$ and edges $x \xrightarrow{\leq 0} 0$ for all other clocks x. All other edges are ∞. We now apply the inclusion test (4) between this distance graph and Z'. Note that is (5) is true, then there is a clock that has $Z'_{x0} < Z_{x0}$. But as $Z_{x0} \leq (\leq, 0)$, we will have $Z'_{x0} < (\leq, 0)$ which implies that $Z'_{x0} < [\![g]\!]_{x0}$. This shows that if the inclusion between zones does not hold, then the inclusion of the guard g in Z' also does not hold. Therefore testing (5) is sufficient. This gives us the following formula with the additional observation that Z'_{x0} can be only lesser than or equal to Z_{x0}.

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } L(w) < d \text{ and} \\ & \exists x.\ \big(Z_{x0} \geq (\leq, -U'_x)\big) \wedge \big((Z'_{x0} < Z_{x0})\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases} \tag{6}$$

Also note that this can be easily extended to an incremental procedure: whenever we add an extra clock to U', then 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 (4) 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 $Z'_{xy} + (<, L_y) < Z_{x0}$. Note that this also entails $[\![g]\!] \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. This is because when g is $w \lessdot d$, we have $[\![g]\!]_{xy} = (<, \infty)$ and $[\![g]\!]_{x0} = (\leq, 0)$ and hence (4) becomes true when Z is substituted with $[\![g]\!]$. Therefore it is sufficient to check (4) for non zero variables

x and y. This gives the following formula function:

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } U(w) < d \text{ and } \exists x, y. \text{ such that} \\ & Z_{x0} \geq (\leq, -U'_x) \text{ and } \big(Z'_{xy} < Z_{xy}\big) \text{ and} \\ & \quad \big(Z'_{xy} + (<, -L'_y) < Z_{x0}\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases} \tag{7}$$

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'.

Upper bound and reset. Here we consider the case when we have guard and reset at the same time. So we consider transition $Z \Rightarrow_{(w<d),R} Z'$. We will combine the cases above since we will treat this transition as

$$Z \Rightarrow_{b:=0} Z^1 \Rightarrow_{w<d} Z^2 \Rightarrow_R Z^3 \Rightarrow_{b<0} Z^4$$

Suppose we have $L'U' = L^4U^4$ that we want to propagate it back to Z. Since b is a clock introduced for technical reasons we can assume that $L^4(b) = U^4(b) = -\infty$. We need to calculate the values of changed edges in all the zones

- In Z^1 we get $Z^1_{b0} = 0$, and $Z^1_{xb} = Z_{x0}$, and $Z^1_{bx} = \infty$.
- In Z^2 we get $Z^2_{xy} = Z^1_{x0} + d + Z_{wy}$ (if this edge changes).
- In Z^3 every edge stays the same but for the clocks that are reset. We have $Z^3_{v0} = 0$, $Z^3_{xv} = Z_{x0}$, and $Z^3_{vx} = \infty$ for $v \in R$ and $x \notin R$.
- In Z^4 we get $Z^4_{xy} = Z_{x0} + Z^2_{by}$ if this edge changes.
  - Suppose $x \notin R$. From the second item we know that $Z^2_{by} = (d + Z_{wy})$. So $Z^4_{xy} = Z_{x0} + d + Z_{wy} = Z^2_{xy}$. This means that no edge changes from Z^3 to Z^4.
  - Suppose $x \in R$ then $Z^4_{xy} = Z^3_{by} = Z^2_{by} = d + Z_{wy}$. Since $Z^3_{xy} = \infty$ this edge necessarily changes.

Because of the last item we see that we always take the guard $x < b$ into U. So $L^3U^3 = L^4U^4[U_b = 0]$. Now $L^2U^2 = L^3U^3[R = -\infty]$. In order to get L^1U^1 we apply the formula (7) using the knowledge what is the relation between $L'U'$ and L^2U^2:

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } U(w) < d \text{ and } \exists x, y \notin R. \text{ such that} \\ & Z^1_{x0} \geq (\leq, -U^3_x) \text{ and } \big(Z^2_{xy} < Z^1_{xy}\big) \text{ and} \\ & \quad \big(Z^2_{xy} + (<, -L'_y) < Z^1_{x0}\big) \\ \max(LU, L'U', L_gU_g) & \exists y \notin R. \text{ such that} \\ & \quad \big(d + Z_{wy} + (<, -L'_y) < 0\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases} \tag{8}$$

The second formula is the specialization of the first for the case of $x = b$. So we see that we almost always take the $w < d$ guard. Observe that the first condition implies the second since $Z^2_{xy} = Z^1_{0x} + d + Z_{wy}$. So if $Z^2_{xy} + (<, -L'_y) < Z^1_{x0}$ then $Z^1_{0x} + d + Z_{wy} + (< -L'_y) < Z^1_{x0}$ which is equivalent to $d + Z_{wy} + (< -L'_y) < Z^1_{x0} - Z^1_{0x}$. But $Z^1_{x0} - Z^1_{0x} \leq 0$ since the zone is not empty.

5.4 Implementation of the *newbounds* function

We consider a transition of the form

$$(q, Z, LU) \xrightarrow{g} (q', Z', L'U')$$

We suppose that `newbounds` function examines this transition. The bounds $L'U'$ have been updated and now we determine how to update the bounds LU. Let X'_L be the set of clocks for which L' bound has been updated. Similarly X'_U for U' bounds.

We will define the new bounds for (q, Z). So the node (q, Z, LU) will be changed to $(q, Z, L_{new}U_{new})$. Observe that the bounds can only increase.

We have four cases depending on the type of the guard. The pseudocode is presented in Algorithm 1.2

Lower bound guard We consider a transition for the form $(q, Z, LU) \xrightarrow{g_L} (q', Z', L'U')$ with $g_l \equiv \wedge_{i=1\ldots k} v_i \geq d_i$. First, we set $L_{new}U_{new}$ to the maximum of LU and $L'U'$; notice that by the defintion of X'_L and X'_U we need to calculate maximum only for the clocks in these two sets. Then we establish the set of edges E of the zone Z' that have changed, and that are relevant for the test (6). The final loop decides which constraints should be taken to increase L bound. We take d_i when it indeed determines some relevant edge from E. If we take d_i then we update L_{new}, and remove from E all edges that are set by d_i. This is because there may be another constraint that influences the same change in Z' and there is no point of taking it.

For the correctness proof let g^1_L be the set of constraints that have been taken and g^2_L the constraints that have been omited. The transition $(q, Z, L_{new}U_{new}) \Rightarrow_{g_L} (q', Z', L'U')$ can be decomposed into $(q, Z, L_{new}U_{new}) \Rightarrow_{g^1_L} (q', Z^1, L'U') \Rightarrow_{g^2_L} (q', Z', L'U')$. From the algorithm we know that all the edges from E as in line 18 are the same in Z^1 and Z'. Hence by formula (6) we get $\mathrm{Post}_{g^2_L}(\mathfrak{a}_{\preccurlyeq L'U'}(Z^1)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. Since all the guards from g^1_L are taken we get $\mathrm{Post}_{g^1_L}(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z^1)$.

Upper bound guard We consider a transition of the form $(q, Z, LU) \xrightarrow{g_U} (q', Z', L'U')$ with $g_l \equiv \wedge_{i=1\ldots k} w_i \leq e_i$. Let us explain Algorithm 1.2 in this case. As in the previous case we set $L_{new}U_{new}$ to the maximum of LU and $L'U'$. Next we calculate the set of edges E that can influence taking a guard. The final for loop considers a constraint one by one. When the constraint implies an edge in E we take the constraing and remove all the edges implied by it.

The correctness proof is very similar to the previous case. Let g^1_U be the set of constarints that have been taken and g^2_U the constraints that have been

Algorithm 1.2. `disabled` and `newbounds` functions

```
1
2  function disabled(q,Z)
3    L:=L_{-∞}; U:=U_{-∞};
4    for every transition t from q disabled from (q,Z) do
5       choose an atomic guard x ≪ d from the guard of t
6       such that Z ⊭ x ≪ d // guard of t has only upper bound guards
7       U(x):=max(d,U(x))
8    return (L,U)
9
10
11
12 function newbounds(v,v,X'_L,X'_U)
13 for every clock x do
14   if x ∈ X'_L then L_new(x):=max(L(x),L'(x)) else L_new(x):=L(x);
15   if x ∈ X'_U then U_new(x):=max(U(x),U'(x)) else U_new(x):=U(x);
16
17 if transtion v → v' is a lower bound guard ⋀_{i=1...k} v_i ≥ d_i
18   E:={(x,0) : x ∈ X'_U and Z_{x0} ≥ (≤,-U'_x) and Z'_{x0} < Z_{x0}}
19   while E ≠ ∅ do
20      choose d_i such that there is (x,0) ∈ E with -d_i + Z_{xv_i} = Z'_{x0};
21      L_new(v_i):=max(d_i,L_new(v_i));
22      E:=E \ {(x,0) : d_i + Z_{xv_i} = Z'_{x0}}
23
24 else if transtion v → v' is an upper bound guard ⋀_{i=1...k} w_i ≤ e_i
25   E:={(x,y) : x ∈ X'_U and y ∈ X'_L, and
26                Z_{x0} ≥ (≤,-U'_x) and Z'_{xy} < Z_{xy} and Z'_{xy} + (<,-L_y) < Z_{x0}};
27   while E ≠ ∅ do
28      choose e_i such that there is (x,y) ∈ E with e_i + Z_{w_i y} + Z_{x0} = Z'_{xy}
29      U_new(w_i):=max(e_i,U_new(w_i));
30      E:=E \ {(x,y) : e_i + Z_{w_i y} + Z_{x0} = Z'_{xy}}
31
32 else if transtion v → v' is a reset R
33   for x ∈ R do
34    L_new(x)=L(x); U_new(x):=U(x);
35
36 else if transition v → v' is an upper bound guard ⋀_{i=1...k} w_i ≤ e_i and
37                                  a reset R
38   Fix some r ∈ R;
39   E:={(r,y) : y ∈ X'_L \ R and Z'_{ry} < (<,L'_y)};
40   while E ≠ ∅ do
41      choose e_i such that there is (r,y) ∈ E with e_i + Z_{wy} = Z'_{ry}
42      U_new(w_i):=max(e_i,U_new(w_i));
43      E:=E \ {(r,y) : e_i + Z_{wy} = Z'_{ry}};
44
45 return(L_new,U_new);
```

omited. The transition $(q, Z, L_{new}U_{new}) \Rightarrow_{g_U} (q', Z', L'U')$ can be decomposed into $(q, Z, L_{new}U_{new}) \Rightarrow_{g_U^1} (q', Z^1, L'U') \Rightarrow_{g_U^2} (q', Z', L'U')$. From the algorithm we know that all the edges from E as in line 25 are the same in Z^1 and Z'. Hence by formula (7) we get $\mathrm{Post}_{g_L^2}(\mathfrak{a}_{\preccurlyeq L'U'}(Z^1)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. Since all the guards from g_L^1 are taken we get $\mathrm{Post}_{g_L^1}(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z^1)$.

Reset The case of reset follows directly from the formula in Section 5.1.

Upped bound and reset This case follows directly from the formula (8).

6 Examples

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

6.1 All edges enabled

Consider the automaton $\mathcal{A}_1$ shown in Figure 3. In the same figure, the zone graph of $\mathcal{A}_1$ has been depicted. Note that the zone graph has no edges disabled and hence is isomorphic to the automaton. In such a case, observe that it is safe to abstract all the zones by the true zone. The set of reachable states of the automaton remain the same even after abstracting all zones to the true zones.

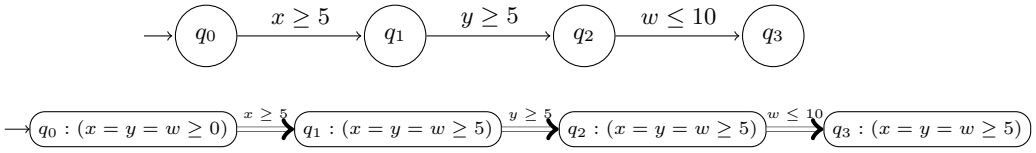

Fig. 3. $\mathcal{A}_1$: all edges enabled in the zone graph

Algorithm 1.1 is able to incorporate this phenomenon. Initially all the constants are $-\infty$ and hence the $\mathfrak{a}_{\preccurlyeq LU}$ abstraction of each zone would give the true zone. The algorithm starts propagating finite LU-constants only when it encounters a disabled edge during exploration. In particular, if there are no edges disabled, all the constants are kept $-\infty$. We will now see an example where this property of the propagation yields exponential gain over the static analysis method and the on-the-fly constant propagation procedure.

Consider the automaton $\mathcal{D}_n$ shown in Figure 4. This is slightly modified from the example given in [LNZ05]. We have changed all guards to check for an equality. It is a parallel composition of three components. Automaton $\mathcal{D}_n$ has $2n$ clocks: $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$. 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 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 $i, j, k \in \{0, \ldots, n\}$. In all, there are

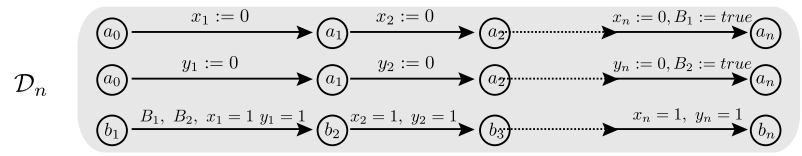

Fig. 4. Automaton $\mathcal{D}_n$

$(n+1)^2 + n$ states in the product automaton. Let us assume that no state is accepting so that any algorithm that explores this automaton should explore the entire zone graph.

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, the zone graph of $\mathcal{D}_n$ should have no edges disabled which implies that the LU-constants given by Algorithm 1.1 in each node are $-\infty$. The number of uncovered nodes in the ASG obtained would be the same as the number of states.

Static analysis: However, the static analysis procedure would give $L = U = 1$ for every clock. We will now see that this would yield a zone graph with at least 2^n nodes.

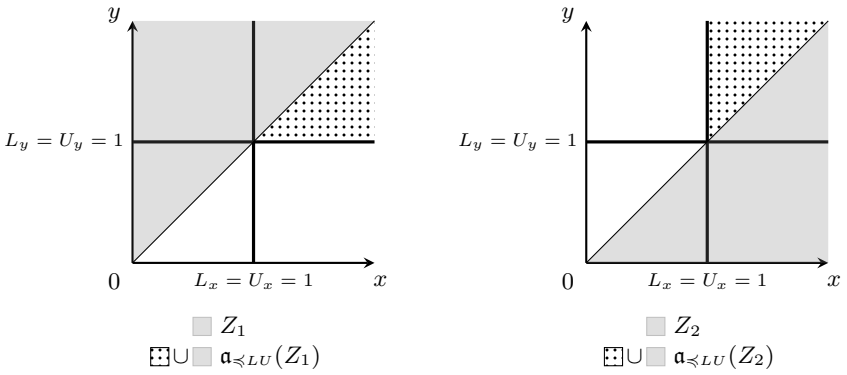

Fig. 5. Zones indistinguishable by $\mathfrak{a}_{\preccurlyeq LU}$

Consider Figure 5 that shows two zones Z_1 and Z_2 and their $\mathfrak{a}_{\preccurlyeq LU}$ abstractions when $L = U = 1$ for both the clocks x and y. Zone Z_1 is given by all valuations that satisfy $x \leq y$. Similarly zone Z_2 is given by all valuations that satisfy $x \geq y$. Observe that Z_1 and Z_2 are incomparable with respect to $\mathfrak{a}_{\preccurlyeq LU}$, that is, $Z_1 \not\subseteq \mathfrak{a}_{\preccurlyeq LU}(Z_2)$ and $Z_2 \not\subseteq \mathfrak{a}_{\preccurlyeq LU}(Z_1)$.

In our example of the automaton $\mathcal{D}_n$, if in a path, x_1 is reset before y_1 then in the state (a_n, a_n, b_0) we would have a zone that entails $y_1 \leq x_1$. Similarly if y_1 is reset before x_1, then the zone would entail $x_1 \leq y_1$. In each of these paths to (a_n, a_n, b_0) clock x_2 could be reset either before or after y_2 and so on for each x_i. There are at least 2^n paths leading to (a_n, a_n, b_0) each of them giving a different zone depending on the order of resets. Note that two zones are incomparable if a projection onto 2 clocks are incomparable. By the argument in the previous paragraph, each of the mentioned zones would be incomparable with respect to the other. Therefore there are at least 2^n uncovered nodes with state (a_n, a_n, b_0).

$\mathfrak{a}_{\preccurlyeq_{LU}}$*,otf:* As all the edges are enabled, the constant propagation algorithm 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.

6.2 Presence of disabled edges

Consider the automaton $\mathcal{A}_2$ in Figure 6. One can see that the last transition with the upper bound is not fireable. The cause of the edge being disabled is because the value of w in all the valuations of Z_3 is bigger than 1. The cause of this increase is the first lower bound guard $x \geq 5$. At q_1 itself, all the valuations have $w \geq 1$. As w is never reset in the automaton, there is no way w can get lesser than 1 after passing this guard. Note that the guards $y \geq 5$ and $z \geq 100$ do not play a role at all in the edge being disabled. Even if they had not been there, the edge would be disabled.

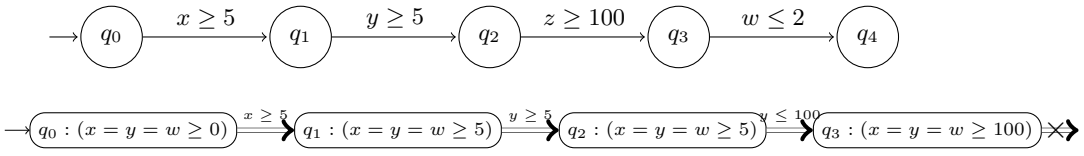

Fig. 6. $\mathcal{A}_2$: One edge disabled

We want to capture this scenario by saying that at q_0 the relevant constants are: $L_0(x) = 5$ and $U_0(x) = 1$ and the rest are $-\infty$. One can verify that Algorithm 1.1 would give exactly these constants. The static analysis algorithm or the constant propagation would give additionally $L(y) = 5$ and $L(z) = 100$, which we have seen are unnecessary. This way, we get smaller constants and hence bigger abstract zones.

We will now see that this pruning can sometimes lead to an exponential gain. We will modify the example $\mathcal{D}_n$ of Section 6.1.

Let $\mathcal{D}'_n$ be the automaton shown in Figure 7. It is the same as $\mathcal{D}_n$ except that now every guard involving y-clock is $y == 2$. Starting from a node

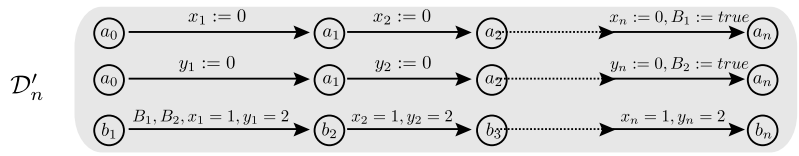

Fig. 7. Automaton $\mathcal{D}'_n$

$((a_n, a_n, b_0), Z, LU)$, it is possible to reach a node with state (a_n, a_n, b_n) only if Z entails $x_i \leq y_i$ for all i. If "fortunately", the order of exploration of the resets leads us to such a zone Z, then this path would yield no constants and hence the abstraction would give the true zone. Due to this there would not be any more exploration from (a_n, a_n, b_0) and we would have the number of uncovered nodes equal to number of states of automaton.

If it is not the case, then there is an i such that $y_i \leq x_i$ and for all $j < i$, $x_i \leq y_i$. Therefore, 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 $L(y_i) = 2$ and propagates these two backwards. These are the relevant guards that cause the disabled edge. Since these are the only constants, in the future, exploration will not occur from a node $((a_n, a_n, b_0), Z', L'U')$ if Z' satisfies $x_i \leq y_i$ as they will be covered. 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 in size quadratic in n.

Static analysis: The static analysis procedure would give $L = U = 2$ for all y-clocks and $L = U = 1$ for all x-clocks. A similar argument as in Section 6.1 would show at least 2^n uncovered nodes with state (a_n, a_n, b_0).

$\mathfrak{a}_{\preccurlyeq_{LU}}$*,otf:* The otf bounds algorithm could work slightly different 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 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

Model	nb. of clocks	UPPAAL (-C) nodes	sec.	$Extra^+_{LU}$,sa nodes	sec.	$\mathfrak{a}_{\preccurlyeq LU}$,otf nodes	sec.	$\mathfrak{a}_{\preccurlyeq LU}$,disabled 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	74324	6.1
CSMA/CD 11	12	311310	5.4	311309	16.8	198669	16.1	188315	15.9
CSMA/CD 12	13	786447	14.8	786446	44.0	493582	41.8	469027	40.9
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	37.6
Fischer 9	9	135485	2.4	135485	8.9	135485	11.4	135485	24.7
Fischer 10	10	447598	10.1	447598	34.0	447598	42.8	447598	98.1
Fischer 11	11	1464971	40.4	1464971	126.8				
Stari 2	7	7870	0.1	6993	0.4	5779	0.4	5113	0.5
Stari 3	10	136632	1.7	113958	9.4	82182	8.2	53178	7.8
Stari 4	13	1323193	26.2	983593	109.0	602762	84.9	342801	65.7

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).

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 the results for the algorithm in [HKSW11] 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 presented in Algorithm 1.1. Similarly to UPPAAL, it uses a Passed/Waiting list instead of a stack. The implemented algorithm is presented in Appendix A.

The results show a huge gain on two examples: $\mathcal{D}''$ and FDDI. D''_n corresponds to the automaton $\mathcal{D}_n$ in Fig. 6 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 last two models, CSMA/CD and Stari [BMT99] show the average situation. The interest of Stari is that it is a very complex example with both a big discrete part and 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 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 [McM06], 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 [MPS11]. We think that the propagation mechanisms presented here are well adapted to such methods.

References

AD94. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

BBFL03. G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In *Proceedings of TACAS*, volume 2619 of *LNCS*, pages 254–270, 2003.

BBLP06. 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.

BDL+06. G. Behrmann, A. David, K. G Larsen, J. Haakansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proceedings of QEST*, pages 125–126. IEEE Computer Society, 2006.

BLR05. P. Bouyer, F. Laroussinie, and P.-A. Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In *FORMATS*, volume 3829 of *LNCS*, pages 112–126, 2005.

BMT99. M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In *CHARME*, volume 1703 of *LNCS*, pages 125–141, 1999.

BY04a. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Proceedings of Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124, 2004.

BY04b. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004.

CY92. C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Form. Methods Syst. Des.*, 1(4):385–415, 1992.

Dil89. D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of AVMFSS*, volume 407 of *LNCS*, pages 197–212, 1989.

DT98. C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Proceedings of TACAS*, volume 1384 of *LNCS*, pages 313–329, 1998.

DWT95. D. L. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *CAV*, volume 939 of *LNCS*, pages 409–422, 1995.

HJMS02. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

HKSW11. F. Herbreteau, D. Kini, B. Srivathsan, and I. Walukiewicz. Using non-convex approximations for efficient analysis of timed automata. In *Proceedings of FSTTCS*, volume 13 of *LIPIcs*, pages 78–89. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

HSW12. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Better abstractions for timed automata. In *LICS*, 2012.

LNZ05. D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theor. Comput. Sci.*, 345(1):27–59, 2005.

LS00. F. Laroussinie and Ph. Schnoebelen. The state-explosion problem from trace to bisimulation equivalence. In *Proceedings of FoSSaCS*, volume 1784 of *LNCS*, pages 192–207, 2000.

McM06. K. McMillan. Lazy abstraction with interpolants. In *CAV*, volume 4144 of *LNCS*, pages 123–136, 2006.

MPS11. G. Morbé, F. Pigorsch, and C. Scholl. Fully symbolic model checking for timed automata. In *Proceedings of CAV*, volume 6806 of *LNCS*, pages 616–632, 2011.

Sor04. M. Sorea. Lazy approximation for dense real-time systems. In *FORMATS/FTRTFT*, volume 3253 of *LNCS*, pages 363–378, 2004.

TAKB96. S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton. Verifying abstractions of timed systems. In *Proceedings of CONCUR*, volume 1119 of *LNCS*, pages 546–562, 1996.

Wan04. Farn Wang. Efficient verification of timed automata with bdd-like data structures. *Int. J. Softw. Tools Technol. Transf.*, 6(1):77–97, 2004.

WT94. H. Wong-Toi. *Symbolic Approximations of Verifying Real-Time Systems*. PhD thesis, Stanford University, November 1994.

A Implementation of Algorithm $\mathfrak{a}_{\preccurlyeq LU}$,disabled

Algorithm 1.3 gives an overview of UPPAAL's algorithm. It takes as input[3] a zone graph and searches for a reachable accepting state. When a new node is expanded (l. 11), it is first checked if it is covered by a visited node (l. 15). If so, then it does not need to be explored. If not, all the nodes that are covered by the new node are removed (l. 20-21) before the new node is inserted to save memory and time.

In order to ensure the termination of the algorithm, the zones are abstracted with an extrapolation operator (e.g. $Extra^{+}_{LU}$[BBLP06]) that guarantees a finite number of abstracted zones. The abstraction parameters are clock bounds LU. They are obtained by a static analysis of the timed automaton[BBFL03].

Algorithm 1.3. UPPAAL's algorithm.

```
1  P := ∅ // Passed list (visited nodes)
2  W := ∅ // Waiting list (W is included in P)
3
4  function main(): // input: zone graph ZG=(v0,V,→)
5    insertPW(v0)
6    while (W is not empty) do
7      pick a node v from W
8      if (v.q is accepting)
9        return ''not empty''
10     for each transition v → v' in ZG do
11       insertPW(v')
12   return ''empty''
13
14 function insertPW(v):
15   if (∃v' ∈ P s.t. v.q = v'.q and v.Z ⊆ v'.Z)
16     // don't add v as it is covered by v'
17     return
18   else
19     // remove all nodes v' covered by v
20     for each v' ∈ P s.t. v'.q = v.q and v'.Z ⊆ v.Z do
21       remove v' from P and from W
```

[3] The implementation builds the zone graph on-the-fly from a timed automaton taken as input.

```
22      // insert v
23      insert v in P and in W
```

Our algorithm $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled is built on top of UPPAAL's algorithm. It is depicted in Algorithm 1.4. The main difference is that it computes dynamic LU-bounds that are used to stop the exploration earlier. The dynamic bounds are used in l. 15. We avoid exploring a node if it is covered by a visited node w.r.t. dynamic bounds and abstraction $\mathfrak{a}_{\preccurlyeq_{LU}}$. If the node is not covered, then its bounds are updated w.r.t. the transitions that are disabled from that node (l. 21) and the node is explored (l. 24).

The algorithm computes an adaptive simulation graph $\rightsquigarrow$ (see Definition 9) and a covering relation $\lhd$. The tentative nodes in Definition 9 are the nodes v that are covered by some node v', that is: $v \lhd v'$. The algorithm propagates the bounds and it updates $\rightsquigarrow$ and $\lhd$ in order to maintain the invariants in Definition 9.

As the bounds are propagated over the graph $\rightsquigarrow$, some covering edge $v' \lhd v$ may become invalid. This is checked in line 50. When the bounds in v' have to be updated from the bounds in the covering node v, it is first checked if v' is still covered by v. If it is not the case, v' is put in the list of waiting nodes and it will be considered again later.

The propagation of clock bounds relies on function `newbounds` given in Algorithm 1.2.

Algorithm 1.4. Algorithm $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled.

```
1  // Assumptions: no lower bound atomic guards d ≤ x in invariants
2  //              no atomic guard x < 0
3
4  P := ∅ // Passed list (visited nodes)
5  W := ∅ // Waiting list (W is included in P)
6  ◁ := ∅ // Covering relation wrt dynamic bounds
7  ⇝ := ∅ // Propagation relation
8
9  function main(): // input: zone graph ZG=(v0,V,→)
10   insertPW(v0)
11   while (W is not empty) do
12     pick a node v from W
13     if (v.q is accepting)
14       return ``not empty''
15     if (∃v' ∈ (P \ W) uncovered st v.q = v'.q and v.Z ⊆ a_{≼v'.LU}(v'.Z))
16       add v ◁ v' and v' ⇝ v
17       v.LU := v'.LU
18       (X_L, X_U) := bounds modified during the copy
19       propagate(v, X_L, X_U)
20     else
21       v.LU := disabled(v)
22       (X_L, X_U) := active clocks in v.LU
23       propagate(v, X_L, X_U)
24       for each transition v → v' in ZG do
```

```
25          add v' ⇝ v
26          insertPW(v')
27    return ''empty''
28
29  function insertPW(v):
30    if (∃v' ∈ P s.t. v.q = v'.q and v.Z ⊆ v'.Z)
31      // v is covered by v' wrt static bounds
32      replace all v ⇝ v'' by v' ⇝ v''
33    else
34      // remove all nodes v' covered by v wrt static bounds
35      for each v' ∈ P s.t. v'.q = v.q and v'.Z ⊆ v.Z do
36        remove v' from P and from W
37        replace all v' ⇝ v'' by v ⇝ v''
38        remove all v'' ⇝ v'
39        if (∃v'' ∈ P st v' ◁ v'')
40          remove v' ◁ v''
41        else
42          for each v'' ∈ P st v'' ◁ v' do
43            remove v'' ◁ v'
44            insert v'' in W
45      insert v in P and in W
46
47  function propagate(v, X_L, X_U):
48    for each v' st v ⇝ v' do
49      if (v' ◁ v) // propagation due to a covering edge
50        if (v'.Z ⊆ 𝔞_{≼v.LU}(v.Z)) // v' still covered by v
51          v'.LU := v.LU
52          (X'_L, X'_U) := bounds modified during the copy
53        else // v' is not covered by v anymore
54          v'.LU := x ↦ −∞; (X'_L, X'_U) := (∅, ∅)
55          insert v' in W
56      else // propagation due to a transition in ZG
57        let t be the transition q' −t→ q that corresponds to v ⇝ v'
58        (g_l, g_u, R) := decompose(t)
59        (L_tU_t, X^t_L, X^t_U) := backwardLU(Z', g_l, g_u, R, v.LU, X_L, X_u)
60        v'.LU := max(v'.LU, L_tU_t)
61        (X'_L, X'_U) := bounds modified by maximization
62      if (X'_L ≠ ∅ or X'_U ≠ ∅)
63        propagate(v', X'_L, X'_U)
64
65  function disabled(v):
66    L := x ↦ −∞; U := x ↦ −∞
67    for each transition t from v.q that is disabled from v.Z do
68      (g_l, g_u, R) := decompose(t) // lower bounds, upper bounds, reset
69      choose an atomic guard w ⋖ d in g_u disabled from v.Z ∧ g_l
70      L_d := x ↦ −∞; U_d := w ↦ d, x ↦ −∞ (x ≠ w)
71      (L_tU_t, X_L, X_u) := backwardLU(v.Z, g_l, true, ∅, L_dU_d, ∅, {x})
72      LU := max(LU, L_tU_t)
73    return LU
```

```
74
75 function decompose(t):
76   let t = (I, g, R, I') // src inv, guard, reset, tgt inv
77   g' := g ∧ I
78   add to g' all the atomic guard x ◁ d from I' st x ∉ R
79   let g'_l be the lower−bound atomic guards d ◁ x in g'
80   let g'_u be the upper−bound atomic guards x ◁ d in g'
81   return (g'_l, g'_u, R)
82
83 function backwardLU(Z, g_l, g_u, R, LU, X_L, X_U):
84   let σ := Z --g_l--> Z' --g_u;R--> Z''
85   update LU, X_L and X_U applying newbounds on Z' --g_u;R--> Z''
86   update LU, X_L and X_U applying newbounds on Z --g_l--> Z'
87   return (LU, X_L, X_u)
```