Lazy Reachability Checking for Timed Automata with Discrete Variables

Tamás Tóth[0000-0001-7176-1278]* and István Majzik[0000-0002-1184-2882]

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

Abstract.
Systems and software with time dependent behavior are often formally specified using timed automata. For practical real-time systems, these specifications typically contain discrete data variables with nontrivial data flow besides real-valued clock variables. In this paper, we propose a lazy abstraction method for the location reachability problem of timed automata that can be used to efficiently control the visibility of discrete variables occurring in such specifications, this way alleviating state space explosion. The proposed abstraction refinement strategy is based on interpolation for variable assignments and symbolic backward search. We combine in a single algorithm our abstraction method with known efficient lazy abstraction algorithms for the handling of clock variables. Our experiments show that the proposed method performs favorably when compared to other lazy methods, and is suitable to significantly reduce the number of states generated during state space exploration.

Keywords: timed automata · model checking · reachability checking · lazy abstraction · visible variables abstraction · zone abstraction · interpolation

1 Introduction

Timed automata [1] is a widely used formalism for the modeling and verification of systems and software with time-dependent behavior. In timed automata models, erroneous or unsafe behavior (that is to be avoided during operation) is often modeled by error locations. The location reachability problem deals with the question whether a given error location is reachable from an initial state along the transitions of the automaton.

As timed automata contain real-valued clock variables, to ensure performance and termination, model checkers for timed automata apply abstraction over clock variables. The standard solution involves performing a forward exploration in the zone abstract domain [7], combined with extrapolation [3] parametrized by bounds appearing in guards, extracted by static analysis [2]. Other zone-based

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

Authors' manuscript. M. del Mar Gallardo, P. Merino (eds.): *Model Checking Software*, LNCS 10869, 2018. The final publication is available at Springer via https://doi.org/10.1007/978-3-319-94111-0_14.

methods propagate bounds lazily for all transitions [11] or along an infeasible path [10], and perform efficient inclusion checking with respect to a non-convex abstraction induced by the bounds [12]. Alternatively, some methods perform lazy abstraction directly over the zone abstract domain [19, 20]. However, in the context of timed automata, methods rarely address the problem of *abstraction for discrete data variables* that often appear in specifications for practical real-time systems, or do so by applying a fully SMT based approach, relying on the efficiency of underlying decision procedures for the abstraction of both continuous and discrete variables.

In our work, we address the location reachability problem of timed automata with discrete variables by proposing an abstraction method that can be used to *lazily control the visibility of discrete variables* occurring in such specifications. If the abstraction is too coarse to disable an infeasible transition, then we propagate the pre-image of the transition backward using weakest precondition computation, and use interpolation (defined for variable assignments) to extract a set of variables that are sufficient to block the transition from the abstract state. We use interpolation in a similar fashion to attempt to enforce coverage of a newly discovered state with an already visited state when possible, this way effectively pruning the search space. Our method does not rely on an interpolating SMT solver, and can be freely combined with zone-based forward search (eager or lazy) methods for efficient handling of clock variables.

We evaluated the proposed abstraction method by combining it with lazy refinement techniques for continuous variables. Results show that in terms of execution time our method performs similarly to lazy methods without abstraction of discrete variables, but generates a smaller (in cases significantly smaller) state space.

Comparison to related work. Lazy abstraction [9], a form of counterexample-guided abstraction refinement [6], is an approach widely used for reachability 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. For timed automata, a lazy abstraction approach based on non-convex LU-abstraction and on-the-fly propagation of bounds has been proposed [10]. 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 or ASG). As a consequence it is able to apply refinement as soon as the abstraction admits a transition disabled in the concrete system. Similar abstraction techniques based on building an ASG include difference bound constraint abstraction [20] and the zone interpolation-based technique of [19]. In our work, we follow the same approach, but for discrete variables instead of clock variables. The proposed abstraction method is orthogonal to the aforementioned techniques and can be freely combined with any of them.

Symbolic handling of integer variables for timed automata is often supported by unbounded fully symbolic SMT-based approaches. Symbolic backward search techniques like [5] and [17] are based on the computation and satisfiability checking of pre-images. In [13], reachability checking for timed automata is addressed by solving Horn clauses. In the IC3-based technique of [15], the problem of discrete variables is not addressed directly, but the possibility of generalization over discrete variables is (to some extent) inherent to the technique. In [14], also based on IC3, generalization of counterexamples to induction is addressed for both discrete and clock variables by zone-based pre-image computation. In our work, we propose an abstraction method over discrete variables that is completely theory agnostic, and does not rely on an SMT-solver.

In [8], an abstraction refinement algorithm is proposed for timed automata that handles clock and discrete variables in a uniform way. There, given a set of visible variables, an abstracted timed automaton is derived from the original by removing all assignments to abstracted variables, and by replacing all constraints by the strongest constraint that is implied and that does not contain abstracted variables. In case the model checker finds an abstract counterexample, a linear test automaton is constructed for the path, which is then composed with the original system to check whether the counterexample is spurious. If the final location of the test automaton is unreachable, a set of relevant variables is extracted from the disabled transition that will be included in the next iteration of the abstraction refinement loop. In our work, we use a similar approach, but instead of building abstractions globally on the system level and then calling to a model checker for both model checking and counterexample analysis, we use a more integrated, lazy abstraction method, where the abstraction is built on-the-fly, and refinement is performed locally in the state space where more precision is necessary.

Interpolation for variable assignments was first described in [4]. There, the interpolant is computed for a prefix and a suffix of a constraint sequence, and an inductive sequence of interpolants is computed by propagating interpolants forward using the abstract post-image operator. In our work, we define interpolation for a variable assignment and a formula, and compute inductive sequences of interpolants by propagating interpolants backward using weakest precondition computation. In our context, this enables us to consider a suffix of an infeasible path, instead of the whole path, for computing inductive sequences of interpolants.

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 the visibility of discrete variables for timed automata. Section 4 describes experiments performed on the proposed algorithm. Finally, conclusions are given in Section 5.

2 Background and Notations

Let V be a set of *data variables* over $\mathbb{Z}$, and X a set of *clock variables* over $\mathbb{R}_{\geq 0}$. A *data constraint* over V is a well-formed formula $\varphi \in DC(V)$ built from variables in V and arbitrary function and predicate symbols interpreted over $\mathbb{Z}$. A *clock constraint* over X is a formula $\varphi \in CC(X)$ that is a conjunction of atoms of the form $x \prec c$ and $x_i - x_j \prec c$ where $x, x_i, x_j \in X$, $c \in \mathbb{Z}$ and $\prec\ \in \{<, \leq\}$. A *data update* over V is an assignment $u \in DU(V)$ of the form $v \coloneqq t$ where $v \in V$ and t is a term built from variables in V and function symbols interpreted over $\mathbb{Z}$. A *clock update* (clock reset) over X is an assignment $u \in CU(X)$ of the form $x \coloneqq n$ where $x \in X$ and $n \in \mathbb{Z}$. The set of variables appearing in a formula φ is denoted by $\mathsf{vars}(\varphi)$.

A *valuation* over a finite set of variables is a function that maps variables to their respective domains. A *data valuation* is a valuation over a set of data variables V, that is, a function $\nu : V \to \mathbb{Z}$. Similarly, a *clock valuation* is a valuation over a set of clock variables X, that is, a function $\eta : X \to \mathbb{R}_{\geq 0}$. We will denote by $Eval(Q)$ the set of valuations over a set of variables Q.

Throughout the paper we will allow partial functions as valuations. We extend valuations to range over terms and formulas the usual way, with the possibility that the value of a term is undefined over a valuation. We will denote by $\sigma \models \varphi$ iff formula φ is satisfied under valuation σ. Note that in the context of partial valuations $\sigma \models \neg\varphi$ is a strictly stronger statement than $\sigma \not\models \varphi$ (e.g. $\{x \hookleftarrow 1\} \not\models y \doteq 1$ but it is not the case that $\{x \hookleftarrow 1\} \models y \not\doteq 1$).

We will denote by $\mathsf{def}(\sigma)$ the domain of definition of a valuation, that is, $\mathsf{def}(\sigma) = \{q \mid \sigma(q) \neq \bot\}$, and by $\mathsf{form}(\sigma)$ the formula characterizing the valuation, that is, $\mathsf{form}(\sigma) = \bigwedge_{q \in \mathsf{def}(\sigma)} q \doteq \sigma(q)$. Valuation $\top$ is the unique valuation such that $\mathsf{def}(\top) = \emptyset$. We denote by $\sigma \sqsubseteq \sigma'$ iff $\sigma(q) = \sigma'(q)$ for all $q \in \mathsf{def}(\sigma')$. Note that $\sqsubseteq$ is a partial order, as expected. Moreover if $\sigma \sqsubseteq \sigma'$ and $\sigma' \models \varphi$ then $\sigma \models \varphi$, and $\sigma \sqsubseteq \sigma'$ iff $\sigma \models \mathsf{form}(\sigma')$.

We will denote by $\otimes$ the partial function over valuations that is defined as

$$(\sigma \otimes \sigma')(q) = \begin{cases} \sigma(q) & \text{if } q \in \mathsf{def}(\sigma) \\ \sigma'(q) & \text{if } q \in \mathsf{def}(\sigma') \\ \bot & \text{otherwise} \end{cases}$$

if $\sigma(q) = \sigma'(q)$ for all $q \in \mathsf{def}(\sigma) \cap \mathsf{def}(\sigma')$, and is undefined otherwise.

Given a valuation $\sigma \in Eval(Q)$ and an assignment $q \coloneqq t$, we denote by $\sigma\{q \coloneqq t\}$ the valuation $\sigma' \in Eval(Q \cup \{q\})$ such that $\sigma'(q) = \sigma(t)$ and $\sigma'(q') = \sigma(q')$ for all $q' \neq q$. For a sequence of updates μ and a set of updates U we define

$$\sigma\{\mu\}_U = \begin{cases} \sigma & \text{if } \mu = \epsilon \\ \sigma\{u\}\{\mu'\}_U & \text{if } \mu = u \cdot \mu' \text{ and } u \in U \\ \sigma\{\mu'\}_U & \text{if } \mu = u \cdot \mu' \text{ and } u \notin U \end{cases}$$

2.1 Timed automata

In the area of real-time verification, timed automata [1] is the most prominent formalism. To make the specification of practical systems more convenient, the traditional formalism is often extended with various syntactic and semantic constructs, in particular with the handling of discrete variables. In the following, we describe such an extension.

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

- L *is a finite set of locations,*
- V *is a finite set of data variables of integer type,*
- X *is a finite set of clock variables,*
- $T \subseteq L \times \mathcal{P}(C) \times U^* \times L$ *is a finite set of transitions with sets* C *and* U *defined as* $C = DC(V) \cup CC(X)$ *and* $U = DU(V) \cup CU(X)$*, where for a transition* (ℓ, G, μ, ℓ')*, the set* $G \subseteq C$ *is a set of guards and* $\mu \in U^*$ *is a sequence of updates,*
- $\ell_0 \in L$ *is the initial location.*

Throughout the paper, we will refer to a timed automaton with discrete variables simply as a timed automaton.

A state of $\mathcal{A}$ is a triple (ℓ, ν, η) where $\ell \in L$, $\nu \in \mathit{Eval}(V)$ and $\eta \in \mathit{Eval}(X)$. We will denote by ν_0 the unique total function $\nu_0 : V \to \{0\}$ and by η_0 the unique total function $\eta_0 : X \to \{0\}$.

Definition 2 (Semantics). *The operational semantics of a timed automaton is given by a labeled transition system with initial state* (ℓ_0, ν_0, η_0) *and two kinds of transitions:*

- Delay: $(\ell, \nu, \eta) \xrightarrow{\delta} (\ell, \nu, \eta')$ *for some real number* $\delta \geq 0$ *where* $\eta' = \eta + \delta$ *with* $(\eta + \delta)(x) = \eta(x) + \delta$ *for all* $x \in X$*;*
- Action: $(\ell, \nu, \eta) \xrightarrow{t} (\ell', \nu', \eta')$ *for some transition* $t = (\ell, G, \mu, \ell')$ *where we have* $\nu' = \mathsf{dpost}_t(\nu)$ *and* $\eta' = \mathsf{cpost}_t(\eta)$ *with partial functions*

$$\mathsf{dpost}_t(\nu) = \begin{cases} \bot & \textit{if } \nu \models \neg g \textit{ for some } g \in G \cap DC(V) \\ \nu\{\mu\}_{DU(V)} & \textit{otherwise} \end{cases}$$

$$\mathsf{cpost}_t(\eta) = \begin{cases} \bot & \textit{if } \eta \models \neg g \textit{ for some } g \in G \cap CC(X) \\ \eta\{\mu\}_{CU(X)} & \textit{otherwise} \end{cases}$$

Here, $\mathsf{dpost}_t(\nu)$ denotes the strongest (discrete) postcondition of ν with respect to transition t. Note that for any $t \in T$, function dpost_t is monotonic with respect to $\sqsubseteq$, as expected. Moreover, we define the weakest (discrete) precondition $\mathsf{wp}_t(\varphi)$ as the formula such that $\nu \models \mathsf{wp}_t(\varphi)$ iff $\mathsf{dpost}_t(\nu) \models \varphi$ for all ν and φ, with respect to t.

A *run* of a timed automaton is a sequence of states from the initial state along the transition relation

$$(\ell_0, \nu_0, \eta_0) \xrightarrow{\alpha_1} (\ell_1, \nu_1, \eta_1) \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} (\ell_n, \nu_n, \eta_n)$$

where $\alpha_i \in T \cup \mathbb{R}_{\geq 0}$ 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 is the solution set of a clock constraint $\varphi \in CC(X)$. For sets of clock valuations Z and Z', we will denote by $Z \sqsubseteq Z'$ iff $Z \subseteq Z'$. Moreover, if Z is a zone and $t \in T$, then

- $\perp = \emptyset$,
- $Z_0 = \{\eta \mid \eta = \eta_0 + \delta \text{ for some } \delta \geq 0\}$ and
- $\mathsf{zpost}_t(Z) = \left\{\eta' \mid (\cdot, \cdot, \eta) \xrightarrow{t} s \xrightarrow{\delta} (\cdot, \cdot, \eta') \text{ for some } \eta \in Z \text{ and } \delta \geq 0\right\}$

are also zones. Here, $\mathsf{zpost}_t(Z)$ represents the strongest postcondition of Z with respect to a transition t of a timed automaton. As defined above, function zpost_t is monotonic with respect to $\sqsubseteq$ for any $t \in T$.

Definition 3 (Symbolic semantics). *The symbolic semantics of a timed automaton is given by a labeled transition system with states of the form (ℓ, ν, Z), with initial state (ℓ_0, ν_0, Z_0), and for $t = (\ell, \cdot, \cdot, \ell')$ with transitions of the form $(\ell, \nu, Z) \overset{t}{\Rightarrow} (\ell', \mathsf{dpost}_t(\nu), \mathsf{zpost}_t(Z))$.*

We will say that a transition t is enabled from a symbolic state (ℓ, ν, Z) iff $(\ell, \nu, Z) \overset{t}{\Rightarrow} (\ell', \nu', Z')$ for some ℓ', ν' and $Z' \neq \perp$, otherwise it is disabled. Note that a transition $t = (\ell, \cdot, \cdot, \cdot)$ is disabled from a symbolic state (ℓ, ν, Z) iff $\mathsf{dpost}_t(\nu) = \perp$ or $\mathsf{zpost}_t(Z) = \perp$.

Definition 4 (Symbolic run). *A symbolic run of a timed automaton is a sequence $(\ell_0, \nu_0, Z_0) \overset{t_1}{\Rightarrow} (\ell_1, \nu_1, Z_1) \overset{t_2}{\Rightarrow} \dots \overset{t_n}{\Rightarrow} (\ell_n, \nu_n, Z_n)$ where $Z_n \neq \perp$.*

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

3 Algorithm for Lazy Reachability Checking

In this section, we present our algorithm for lazy reachability checking of timed automata with discrete variables. During the description, we will focus on the handling of discrete variables, but formulate the algorithm so that it is straightforward to combine the method with a corresponding (eager or lazy) method for the handling of clock variables.

3.1 Adaptive simulation graph

The central structure of the algorithm is an abstract simulation graph. The presented formulation is a generalization of the definition presented in [19] for the handling of discrete variables and the possibility of using various methods for the handling of clock variables.

Definition 5 (Unwinding). *An unwinding of a timed automaton* (L, V, X, T, ℓ_0) *is a tuple* $U = (N, E, n_0, M_n, M_e, \rhd)$ *where*

- (N, E) *is a directed tree rooted at node* $n_0 \in N$*,*
- $M_n : N \to L$ *is the node labeling,*
- $M_e : E \to T$ *is the edge labeling and*
- $\rhd \subseteq N \times N$ *is the covering relation.*

For an unwinding we require that the following properties hold:

- $M_n(n_0) = \ell_0$*,*
- *for each edge* $(n, n') \in E$ *the transition* $M_e(n, n') = (\ell, \cdot, \cdot, \ell')$ *is such that* $M_n(n) = \ell$ *and* $M_n(n') = \ell'$*,*
- *for all nodes* n *and* n' *such that* $n \rhd n'$ *it holds that* $M_n(n) = M_n(n')$*.*

The purpose of the covering relation $\rhd$ is to mark that a node of the search tree has been pruned due to another node that admits all runs that are possible from the covered node. We define the following shorthand notations for convenience: $\ell_n = M_n(n)$ and $t_{n,n'} = M_e(n, n')$.

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

- U *is an unwinding of* $\mathcal{A}$*,*
- $\psi_\nu, \psi_{\hat{\nu}} : N \to \mathit{Eval}(V)$ *are labelings of nodes by data valuations and*
- $\psi_Z, \psi_{\hat{Z}} : N \to \mathcal{P}(\mathit{Eval}(X))$ *are labelings of nodes by sets of clock valuations.*

We will use the following shorthand notations: $\nu_n = \psi_\nu(n)$, $\hat{\nu}_n = \psi_{\hat{\nu}}(n)$, $Z_n = \psi_Z(n)$ and $\hat{Z}_n = \psi_{\hat{Z}}(n)$.

A node n is *expanded* iff for all transitions $t \in T$ such that $t = (\ell, \cdot, \cdot, \cdot)$ and $\ell_n = \ell$, either t is disabled from (ℓ_n, ν_n, Z_n), or n has a successor for t. A node n is *covered* iff $n \rhd n'$ for some node n'. It is *excluded* iff it is covered or it has an excluded parent. A node is *complete* iff it is either expanded or excluded. A node n is ℓ-safe iff $\ell_n \neq \ell$.

For an ASG to be useful for reachability checking, we have to introduce restrictions on the labeling. Therefore while building the ASG we will ensure that (ℓ_n, ν_n, Z_n) represents an exact set of reachable states for n (thus with Z_n being a zone), and that $\nu_n \sqsubseteq \hat{\nu}_n$ and $Z_n \sqsubseteq \hat{Z}_n$. We formalize this notion in the next definition.

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

– *(initiation) if* $n = n_0$*, then*
  (a) $\nu_n = \nu_0$ *and* $Z_n = Z_0$
  (b) $\nu_0 \sqsubseteq \hat{\nu}_n$ *and* $Z_0 \sqsubseteq \hat{Z}_n$
– *(consecution) if* $n \neq n_0$*, then for its parent* m *and the transition* $t = t_{m,n}$
  (a) $\nu_n = \mathsf{dpost}_t(\nu_m)$ *and* $Z_n = \mathsf{zpost}_t(Z_m)$
  (b) $\mathsf{dpost}_t(\hat{\nu}_m) \sqsubseteq \hat{\nu}_n$ *and* $\mathsf{zpost}_t(\hat{Z}_m) \sqsubseteq \hat{Z}_n$
– *(coverage) if* $n \rhd n'$ *for some node* n'*, then* $\hat{\nu}_n \sqsubseteq \hat{\nu}_{n'}$ *and* $\hat{Z}_n \sqsubseteq \hat{Z}_{n'}$ *and* n' *is not excluded*
– *(simulation) if* n *is expanded, then any transition disabled from* (ℓ_n, ν_n, Z_n) *is also disabled from* $(\ell_n, \hat{\nu}_n, \hat{Z}_n)$*.*

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. The main challenge for the construction of a well-labeled ASG as defined above is how the labelings $\psi_{\hat{\nu}}$ and $\psi_{\hat{Z}}$ are computed. A well-labeled ASG preserves reachability information, which is expressed by the following proposition.

Proposition 2. *Let* $\mathcal{G}$ *be a complete, well-labeled ASG for a timed automaton* $\mathcal{A}$*. Then* $\mathcal{A}$ *has a symbolic run* $(\ell_0, \nu_0, Z_0) \stackrel{t_1}{\Rightarrow} (\ell_1, \nu_1, Z_1) \stackrel{t_2}{\Rightarrow} \ldots \stackrel{t_k}{\Rightarrow} (\ell_k, \nu_k, Z_k)$ *iff* $\mathcal{G}$ *has a non-excluded node* n *such that* $\ell_k = \ell_n$*.*

Proof. The right-to-left direction is a consequence of the subsequent Lemma 1. and the converse follows from Lemma 2. □

Lemma 1. *Let* $\mathcal{G}$ *be a well-labeled ASG for a timed automaton* $\mathcal{A}$*. If* $\mathcal{G}$ *has a node* n *then* $\mathcal{A}$ *has a symbolic run* $(\ell_0, \nu_0, Z_0) \stackrel{t_1}{\Rightarrow} (\ell_1, \nu_1, Z_1) \stackrel{t_2}{\Rightarrow} \ldots \stackrel{t_k}{\Rightarrow} (\ell_k, \nu_k, Z_k)$ *such that* $\ell_k = \ell_n$*.*

Proof. The statement is a direct consequence of conditions *initiation(a)* and *consecution(a)*. □

Lemma 2. *Let* $\mathcal{G}$ *be a complete, well-labeled ASG for a timed automaton* $\mathcal{A}$*. If* $\mathcal{A}$ *has a symbolic run* $(\ell_0, \nu_0, Z_0) \stackrel{t_1}{\Rightarrow} (\ell_1, \nu_1, Z_1) \stackrel{t_2}{\Rightarrow} \ldots \stackrel{t_k}{\Rightarrow} (\ell_k, \nu_k, Z_k)$ *then* $\mathcal{G}$ *has a non-excluded node* n *such that* $\ell_k = \ell_n$ *and* $\nu_k \sqsubseteq \hat{\nu}_n$ *and* $Z_k \sqsubseteq \hat{Z}_n$*.*

Proof. We prove the statement by induction on the length k of the symbolic run. If $k = 0$, then $\ell = \ell_0$ and $\nu = \nu_0$ and $Z = Z_0$, thus n_0 is a suitable witness by condition *initiation(b)*. Suppose the statement holds for runs of length at most $k - 1$. Hence there exists a non-excluded node m such that $\ell_{k-1} = \ell_m$ and $\nu_{k-1} \sqsubseteq \hat{\nu}_m$ and $Z_{k-1} \sqsubseteq \hat{Z}_m$.

Clearly the transition t_k is not disabled from $(\ell_m, \hat{\nu}_m, \hat{Z}_m)$, as then by the induction hypothesis it would be also disabled from $(\ell_{k-1}, \nu_{k-1}, Z_{k-1})$, which contradicts our assumption. As m is complete and not excluded, it is expanded, and thus has a successor n for transition t_k with $\ell_n = \ell_k$. By condition *consecution(b)*, we have $\mathsf{dpost}_{t_k}(\hat{\nu}_m) \sqsubseteq \hat{\nu}_n$. As $\nu_{k-1} \sqsubseteq \hat{\nu}_m$ and dpost_t is monotonic w.r.t. $\sqsubseteq$, we have $\nu_k \sqsubseteq \hat{\nu}_n$. We can obtain $Z_k \sqsubseteq \hat{Z}_n$ symmetrically.

Thus if n is not covered, then it is a suitable witness for the statement. Otherwise there exists a node n' such that $n \triangleright n'$. By condition *coverage*, we know that $\hat{\nu}_n \sqsubseteq \hat{\nu}_{n'}$ and $\hat{Z}_n \sqsubseteq \hat{Z}_{n'}$ and n' is not excluded, thus n' is a suitable witness. $\square$

3.2 Reachability algorithm

The pseudocode of the algorithm is shown in Algorithm 1. The algorithm gets as input a timed automaton $\mathcal{A}$ and a distinguished error location $\ell_e \in L$. The goal of the algorithm is to decide whether ℓ_e is reachable for $\mathcal{A}$. To this end the algorithm gradually builds an ASG for $\mathcal{A}$ and continually maintains its well-labeledness. Upon termination, it either witnesses reachability of ℓ_e by a node n such that $\ell_n = \ell_e$, which by Lemma 1 corresponds to a symbolic run of $\mathcal{A}$ to ℓ_e, or produces a closed, well-labeled, ℓ_e-safe ASG that proves unreachability of ℓ_e by Lemma 2.

The main data structures of the algorithm are the ASG $\mathcal{G}$ and sets *passed* and *waiting*. The set *passed* is used to store nodes that are expanded and *waiting* s-tores nodes that are incomplete. The algorithm consists of subprocedures CLOSE, EXPAND and REFINE, and of procedures ZCOVER and ZBLOCK. Procedure ZCOVER and ZBLOCK serve for abstraction refinement over clock variables. These procedures can be soundly implemented in various ways [3, 10–12, 19, 20], and we assume such an implementation. Procedure CLOSE attempts to cover a node by some other node. Procedure EXPAND expands a node by creating the successors of a node for all non-blocked transitions for the given location. Procedure REFINE (see in Section 3.3) can be used to ensure for a node n and some formula φ that if $\nu_n \models \varphi$ then $\hat{\nu}_n \models \varphi$ as well. Both CLOSE and EXPAND maintain well-labeledness by calls to REFINE. In particular, CLOSE calls to REFINE in order to enforce condition *coverage*, and EXPAND calls to REFINE to establish condition *simulation*.

The algorithm consists of a single loop in line 8 that employs the following strategy. The loop consumes nodes from *waiting* one by one. If *waiting* becomes empty, then $\mathcal{A}$ is deemed safe. Otherwise, a node n is removed from *waiting*. 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 by a call to CLOSE. If there are no suitable candidates for coverage, then the algorithm establishes completeness of the node by expanding it using EXPAND, which puts it in *passed* and puts all its successors in *waiting*.

We show that EXPLORE is correct with respect to the annotations (procedure contracts) in Algorithm 1. As, given a suitable refinement method for clock variables, termination of the algorithm is trivial, we focus on partial correctness.

Proposition 3. *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). Let $\mathit{covered} = \{n \in N \mid n \text{ is covered}\}$. It is easy to verify that the algorithm maintains the following invariants:

Algorithm 1 Reachability algorithm for timed automata with discrete variables

```
ensure $\rho =$ SAFE iff $\ell_e$ is unreachable for $\mathcal{A}$
function EXPLORE($\mathcal{A}, \ell_e$) returns $\rho \in \{$SAFE, UNSAFE$\}$
    let $n_0$ be a node with $\ell_{n_0} = \ell_0$, $\nu_{n_0} = \nu_0$, $\hat{\nu}_{n_0} = \top$, $Z_{n_0} = Z_0$ and $\hat{Z}_{n_0} = \top$
    $N \leftarrow \{n_0\}$, $E \leftarrow \emptyset$, $\triangleright \leftarrow \emptyset$
    let $\mathcal{G}$ be an ASG for $\mathcal{A}$ over $N$, $E$ and $\triangleright$

    $passed \leftarrow \emptyset$, $waiting \leftarrow \{n_0\}$
    while $n \in waiting$ for some $n$ do
        $waiting \leftarrow waiting \setminus \{n\}$
        if $\ell_n = \ell_e$ then
            return UNSAFE
        else
            CLOSE($n$)
            if $n$ is not covered then
                EXPAND($n$)
    return SAFE

procedure CLOSE($n$)
    for all $n' \in passed$ such that $\ell_n = \ell_{n'}$ and $\nu_n \sqsubseteq \hat{\nu}_{n'}$ and $Z_n \sqsubseteq \hat{Z}_{n'}$ do
        REFINE($n, \mathsf{form}(\hat{\nu}_{n'})$)
        ZCOVER($n, n'$)
        if $\hat{\nu}_n \sqsubseteq \hat{\nu}_{n'}$ and $\hat{Z}_n \sqsubseteq \hat{Z}_{n'}$ then
            $\triangleright \leftarrow \triangleright \cup \{(n, n')\}$
            return

ensure $n$ is expanded
procedure EXPAND($n$)
    for all $t \in T$ such that $t = (\ell, \cdot, \cdot, \ell')$ with $\ell = \ell_n$ do
        let $\nu' = \mathsf{dpost}_t(\nu_n)$
        let $Z' = \mathsf{zpost}_t(Z_n)$
        if $\nu' = \bot$ then
            REFINE($n, \mathsf{wp}_t(\bot)$)
        else if $Z' = \bot$ then
            ZBLOCK($n, t$)
        else
            let $n'$ be a new node with $\ell_{n'} = \ell'$, $\nu_{n'} = \nu'$, $Z_{n'} = Z'$, $\hat{\nu}_{n'} = \top$, $\hat{Z}_{n'} = \top$
            let $(n, n')$ be a new edge with $t_{n,n'} = t$
            $N \leftarrow N \cup \{n'\}$, $E \leftarrow E \cup \{(n, n')\}$
            $waiting \leftarrow waiting \cup \{n'\}$
    $passed \leftarrow passed \cup \{n\}$

require $\nu_n \models \varphi$
ensure $\hat{\nu}_n \models \varphi$
procedure REFINE($n, \varphi$)

require $Z_n \sqsubseteq \hat{Z}_{n'}$
ensure $\hat{Z}_n \sqsubseteq \mathsf{old}(\hat{Z}_{n'})$
procedure ZCOVER($n, n'$)

require $\mathsf{zpost}_t(Z) = \bot$
ensure $\mathsf{zpost}_t(\hat{Z}) = \bot$
procedure ZBLOCK($n, t$)
```

- $N = passed \cup waiting \cup covered$,
- *passed* is a set of non-excluded, expanded, ℓ_e-safe nodes,
- *waiting* is a set of non-excluded, non-expanded nodes,
- *covered* is a set of covered, non-expanded, ℓ_e-safe nodes.

It is easy to see that under the above assumptions sets *passed*, *waiting* and *covered* form a partition of N. Assuming that $\mathcal{G}$ is well-labeled, partial correctness of the algorithm is then a direct consequence. At line 11 a node is encountered that is not ℓ_e-safe, thus by Lemma 1 there is a symbolic run of $\mathcal{A}$ to ℓ_e. Conversely, at line 16 the set *waiting* is empty, so $\mathcal{G}$ is complete and ℓ_e-safe, and as a consequence of Lemma 2 the location ℓ_e is indeed unreachable for $\mathcal{A}$.

What remains to show is that the algorithm maintains well-labeledness. We assume that procedures ZCOVER and ZBLOCK and procedure REFINE maintain well-labeledness (this later statement we prove to hold in Section 3.3). Initially node n_0 is well-labeled as it satisfies *initiation*. Procedure CLOSE trivially maintains well-labeledness, as it just possibly adds a covering edge for two nodes such that condition *coverage* is not violated. For procedure EXPAND, if a given transition t is enabled, then a node is created that satisfies *consecution*. Otherwise the corresponding refinement procedure is called, ensuring that *simulation* holds for the given transition. In particular, if t is blocked due to $\mathsf{dpost}_t(\nu_n) = \bot$, we have $\nu_n \models \mathsf{wp}_t(\bot)$, and thus can call REFINE to update $\hat{\nu}_n$ so that $\hat{\nu}_n \models \mathsf{wp}_t(\bot)$, ensuring $\mathsf{dpost}_t(\hat{\nu}_n) \models \bot$ and effectively disabling t from $(\cdot, \hat{\nu}_n, \cdot)$. □

3.3 Abstraction refinement

To maintain well-labeledness, the algorithm relies on procedure REFINE that performs abstraction refinement by safely adjusting abstract data valuations labeling nodes of the ASG. The pseudocode of the refinement algorithm is shown in Algorithm 2.

Informally, REFINE works as follows. Given a node n and a formula φ such that $\nu_n \models \varphi$ holds, a weakening ν_I of ν_n is computed such that $\nu_I \models \varphi$ by calling to procedure INTERPOLATE, which simply removes variables from the domain of definition that are not necessary for satisfying the formula. Then all covering edges are dropped that would violate condition *coverage* after strengthening. To maintain condition *consecution(b)*, procedure REFINE is then recursively called for the predecessor m of n. The computed interpolant is then used to strengthen the current labeling by including variables occurring in the interpolant in the current abstraction. We show that REFINE maintains well-labeledness and is correct with respect to the annotations in Algorithm 2.

Proposition 4. *Procedure* REFINE *is totally correct: if $\nu_n \models \varphi$, then* REFINE(n, φ) *terminates and ensures $\hat{\nu}_n \models \varphi$. Moreover, it maintains well-labeledness.*

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

Function INTERPOLATE has no side effect, it thus trivially maintains well-labeledness. Moreover, it is easy to see that it satisfies its contract, as it simply

Algorithm 2 Refinement of visible variables

require $\nu_n \models \varphi$
ensure $\hat{\nu}_n \models \varphi$
procedure REFINE(n, φ)
    if $\hat{\nu}_n \models \varphi$ **then**
        return
    else
        let $\nu_I = \text{INTERPOLATE}(\nu_n, \varphi)$
        for all m such that $m \triangleright n$ and $\hat{\nu}_m \not\sqsubseteq \nu_I$ **do**
            $\triangleright \leftarrow \triangleright \setminus (m, n)$
            $\mathit{waiting} \leftarrow \mathit{waiting} \cup \{m\}$
        if $(m, n) \in E$ for some m **then**
            let $t = t_{m,n}$
            $\text{REFINE}(m, \mathsf{wp}_t(\mathsf{form}(\nu_I)))$
        $\hat{\nu}_n \leftarrow \hat{\nu}_n \otimes \nu_I$

require $\nu_A \models \varphi_B$
ensure $\nu_A \sqsubseteq \nu_I$
ensure $\nu_I \models \varphi_B$
ensure $\mathsf{def}(\nu_I) \subseteq \mathsf{def}(\nu_A) \cap \mathsf{vars}(\varphi_B)$
function INTERPOLATE(ν_A, φ_B) **returns** ν_I
    $\nu_I \leftarrow \nu_A|_{\mathsf{vars}(\varphi_B)}$
    let $Q = \mathsf{def}(\nu_A) \cap \mathsf{vars}(\varphi_B)$
    for all $v \in Q$ **do**
        let $\nu'_I = \nu_I|_{\mathsf{def}(\nu_I) \setminus \{v\}}$
        if $\nu'_I \models \varphi_B$ **then**
            $\nu_I \leftarrow \nu'_I$
    return ν_I

drops variables not necessary to ensure satisfiability of φ_B from the domain of definition of ν_A.

In procedure REFINE, if $\hat{\nu}_n \models \varphi$ then no refinement is needed, and the contract is trivially satisfied. Otherwise, the interpolant ν_I is computed by function INTERPOLATE. As $\nu_n \sqsubseteq \hat{\nu}_n$ by well-labeledness and $\nu_n \sqsubseteq \nu_I$ by the precondition, we know that $\hat{\nu}_n \otimes \nu_I$, and thus the new value of $\hat{\nu}_n$, is defined. As $\hat{\nu}_n \otimes \nu_I \sqsubseteq \nu_I$ and $\nu_I \models \varphi$, we have $\hat{\nu}_n \otimes \nu_I \models \varphi$, which ensures the postcondition.

Next we show that well-labeledness is maintained. Condition *simulation* is trivially ensured, as if $\hat{\nu}_n \models \neg g$ for some guard g, then $\hat{\nu}_n \otimes \nu_I \models \neg g$ as well. After the loop we have $\hat{\nu}_m \sqsubseteq \nu_I$ for all m such that $m \triangleright n$. Moreover, $\hat{\nu}_m \sqsubseteq \hat{\nu}_n$ by well-labeledness. Thus $\hat{\nu}_m \sqsubseteq \hat{\nu}_n \otimes \nu_I$, which ensures condition *coverage*. If n has no parent then condition *initiation(b)* is trivially maintained. Otherwise we have $\nu_n \sqsubseteq \nu_I$, thus $\mathsf{dpost}_t(\nu_m) \models \mathsf{form}(\nu_I)$, from which $\nu_m \models \mathsf{wp}_t(\mathsf{form}(\nu_I))$ follows. Hence REFINE can be called to ensure $\hat{\nu}_m \models \mathsf{wp}_t(\mathsf{form}(\nu_I))$, and thus $\mathsf{dpost}_t(\hat{\nu}_m) \sqsubseteq \nu_I$. Moreover, $\mathsf{dpost}_t(\hat{\nu}_m) \sqsubseteq \hat{\nu}_n$ by well-labeledness. It follows that $\mathsf{dpost}_t(\hat{\nu}_m) \sqsubseteq \hat{\nu}_n \otimes \nu_I$, which ensures condition *consecution(b)*. $\square$

3.4 Example

In this subsection, we give an example that demonstrates how the algorithm described above lazily controls the visibility of discrete variables of the system during construction of the abstraction.

Figure 1 shows automaton $\mathcal{A}_k$, a modified version of the examples given in [10,16] where clock variables are replaced by discrete variables and a component is added that nondeterministically increments all variables. The resulting automaton is the parallel composition of four components, and has $2k$ discrete variables, namely $a_1, a_2, \ldots, a_k$ and $b_1, b_2, \ldots, b_k$.

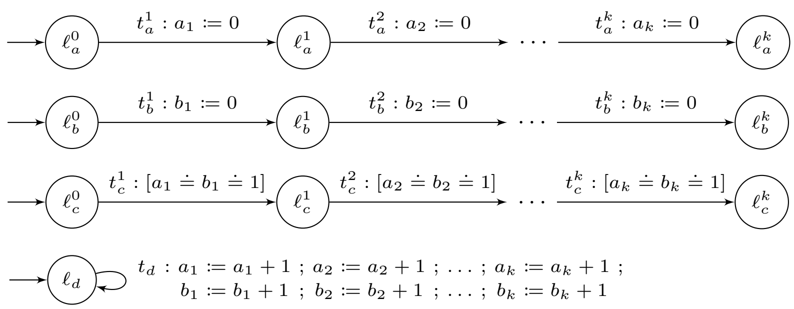

Fig. 1. Automaton $\mathcal{A}_k$

As an example, we are going to consider $\mathcal{A}_1$, the simplest version of the automaton. For simplicity, we are going to omit the indexes in names whenever possible. Figure 2 shows part of the ASG produced by the algorithm. Here, normal edges represent edges of the unwinding (elements of the relation E), dashed edges represent covering edges (elements of the relation $\triangleright$), and dotted edges represent edges of the unwinding that lead to subtrees omitted from the figure. For each node n, the set of visible variables $\mathsf{def}(\hat{\nu}_n)$ is shown.

The algorithm starts by instantiating the root node n_0 with $\hat{\nu}_{n_0} = \top$. As transition t_c is disabled from ν_{n_0} but not from $\hat{\nu}_{n_0}$, the set of visible variables has to be refined in n_0. Hence during refinement, a will be included in the set of visible variables, ensuring $\hat{\nu}_{n_0} = \{a \leftarrow\!\shortmid 0\} \models (a \not\doteq 1 \vee b \not\doteq 1) = \mathsf{wp}_{t_c}(\bot)$. For the same reason, a will become visible when expanding n_1 and n_2. For any other node n however, t_c is either not an outgoing transition of location ℓ_n, or is enabled from ν_n, thus no refinement will be triggered during expansion, resulting in abstraction $\hat{\nu}_n = \top$. This enables coverage between nodes that assign different concrete values to the variables. E.g. covering edges (n_5, n_4) and (n_{10}, n_9) are only possible because b is not visible in either nodes (as $\nu_{n_4} = \nu_{n_9} = \{a \leftarrow\!\shortmid 1, b \leftarrow\!\shortmid 1\}$ and $\nu_{n_5} = \nu_{n_{10}} = \{a \leftarrow\!\shortmid 1, b \leftarrow\!\shortmid 0\}$). More importantly, the algorithm is able to quickly cover nodes that result from the second firing of t_d along a path, thus the resulting ASG remains finite. Even if the number of times t_d can be taken is

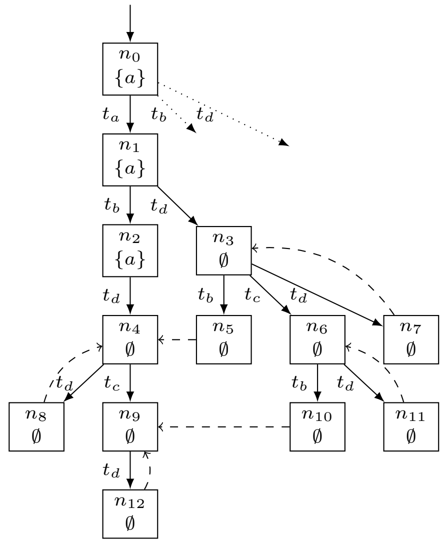

Fig. 2. ASG of $\mathcal{A}_1$

bounded by some number N, an algorithm that handles discrete variables explicitly would generate a significantly larger state space depending on N. Similarly, as k increases, the advantage of the abstraction based method compared to the explicit handling of variables becomes increasingly notable.

4 Evaluation

We implemented a prototype version of our algorithm in the open source model checking framework Theta [18]. In order to enable abstraction refinement for clock variables, we implemented a variant of the lazy abstraction method of [10] based on LU-bounds, and the method described in [19] based on interpolation for zones (with refinement strategy SEQ). These strategies are then combined both with the explicit handling of discrete variables, resulting in algorithms similar to that of the original papers [10, 19], and with the abstraction and refinement method proposed in this paper. The algorithms are evaluated for both breadth-first and depth-first search orders. This results in 8 algorithm configurations by combining the above mentioned alternatives:

- explicit (E) or abstraction-based (A) handling of discrete variables,
- lazy $\mathbf{a}_{\preccurlyeq LU}$ abstraction (L) or interpolation (I) for clock variables and
- breadth-first (B) or depth-first (D) search order.

For the configurations that handle discrete variables explicitly, we partitioned the set of nodes based on the value of the data valuation, this way saving the

$\mathcal{O}(n)$ cost of checking inclusion for valuations. This optimization also significantly reduces the number of nodes for which coverage is checked and attempted during CLOSE. Apart from this and the difference in refinement strategies, the implementation of the configurations is shared.

As inputs we considered 15 timed automata models in UPPAAL 4.0 XTA format that contain integer variables. For each model, the number of discrete variables / number of clock variables is given in parentheses.

- bocdp (26/3), bocdpf (26/3): models of the Bang & Olufsen Collision Detection Protocol obtained from the UPPAAL[1] benchmark set
- brp (9/7): a model of the Bounded Retransmission Protocol
- c1 (12/3), c2 (14/3), c3 (15/3), c4 (17/3): models of a real-time mutual exclusion protocol obtained from the MCTA[2] benchmark set
- m1 (11/4), m2 (13/4), m3 (13/4), m4 (15/4), n1 (11/7), n2 (13/7), n3 (13/7), n4 (15/7): industrial cases studies obtained from the MCTA benchmark set

We performed our measurements on a machine running Windows 10 with a 2.6GHz dual core CPU and 8GB of RAM. We evaluated the algorithm configurations for both execution time (Table 1) and the number of nodes in the resulting ASG (Table 2). The timeout (denoted by "—" in the tables) was set to 120 seconds. In the tables the best values among both the explicit and abstraction based configurations are emphasized with bold font for each model. The execution time is the average of 10 runs, obtained from 12 deterministic runs by removing the slowest and the fastest one.

As can be seen in Table 1, in general, the performance of the fastest configurations of the two categories (explicit and abstraction based configurations) with respect to execution time is balanced (there are no more difference than 100%). For models c1-3, the explicit configuration was faster, but the absolute difference in execution time is not significant. For the other MCTA models, the fastest configurations perform similarly with respect to execution time. For model bocdpf the abstraction-based variant was almost twice as fast, whereas the opposite is true for models bocdp and brp. In total, the abstraction based variant is faster than the corresponding configuration without abstraction in one fourth of the cases, and configuration AID is faster than a given configuration without abstraction in two thirds of the cases.

When comparing the methods based on the number of ASG nodes generated, the difference is more significant, as it can be seen in Table 2. As expected, the abstraction-based method produces a smaller ASG than the corresponding configuration without abstraction in most (97%) of the cases, and the state space generated by configuration AID is smaller in all cases. On average, the reduction in size in favor of the abstraction based handling of discrete variables is around 50%. In the worst case (model c1), the reduced size is around 80%, and in the best case (model bocdpf) it is 15%, i.e. the introduction of abstraction has significant gain.

[1] https://www.it.uu.se/research/group/darts/uppaal/benchmarks
[2] http://gki.informatik.uni-freiburg.de/tools/mcta

Table 1. Execution time in seconds per model and configuration

	EIB	ELB	EID	ELD	AIB	ALB	AID	ALD
bocdp	11.2	**4.8**	8.7	7.0	11.7	11.1	8.7	**7.9**
bocdpf	23.7	**14.3**	20.0	16.4	14.9	13.4	7.7	**7.5**
brp	12.0	**5.4**	20.9	9.2	12.2	**9.5**	14.3	16.3
c1	2.0	**1.3**	1.6	1.8	3.6	4.0	**2.9**	3.2
c2	5.3	**3.2**	3.9	4.7	7.1	8.5	**5.0**	6.8
c3	6.2	**4.5**	5.0	4.9	8.5	9.1	**6.9**	7.6
c4	71.5	53.9	**43.2**	52.4	59.8	77.2	**41.0**	49.6
m1	2.0	1.8	**0.9**	1.5	2.5	4.6	**1.1**	1.7
m2	4.6	4.7	**2.3**	4.3	6.5	12.4	**2.1**	4.2
m3	5.2	4.7	**2.4**	4.6	7.2	13.0	**2.6**	4.7
m4	17.4	23.1	**6.3**	16.0	27.4	68.5	**6.1**	—
n1	2.4	2.2	**1.2**	1.6	2.8	4.4	**1.2**	1.6
n2	6.2	5.9	**3.0**	4.3	7.0	13.9	**2.6**	4.7
n3	6.1	6.0	**3.4**	4.9	7.7	14.5	**2.8**	4.8
n4	23.9	31.5	**7.5**	27.8	30.5	78.6	**5.6**	18.0

Table 2. Number of nodes in the ASG per model and configuration

	EIB	ELB	EID	ELD	AIB	ALB	AID	ALD
bocdp	94801	**74052**	84136	96133	32639	34107	**29846**	32520
bocdpf	212225	**172865**	182085	196003	38492	39801	**26544**	29491
brp	**72117**	96624	114198	159249	**39702**	68979	52049	104552
c1	20967	**18590**	18612	23030	17155	20825	**14973**	18155
c2	67433	67325	**57260**	70198	44711	58351	**39644**	47725
c3	86285	85695	**76122**	94887	50617	62215	**46594**	55473
c4	876266	866890	**737271**	917527	339560	418619	**318470**	384214
m1	8541	19217	**3650**	14720	4394	13078	**1941**	4868
m2	31932	73667	**15610**	62879	16246	39773	**5728**	15797
m3	38128	74514	**15966**	73879	18463	42574	**6707**	17783
m4	145378	297343	**63523**	250221	66406	146804	**20519**	—
n1	7510	18660	**3915**	13132	4222	11802	**1942**	4222
n2	32038	79741	**15534**	54954	15819	42937	**5932**	17695
n3	32799	83982	**16602**	68010	17014	44741	**6547**	17903
n4	142053	325485	**60120**	342408	64934	155729	**17568**	70762

To characterize the fastest configurations, Figure 3 depicts the execution time (first column in blue) and number of nodes generated (second column in red) for the fastest configuration with abstraction relative to the performance of the fastest configuration without abstraction. Similarly, Figure 4 depicts the relative performance when considering the configurations generating the least number of nodes. According to Figure 3, if the configuration with abstraction performs well in execution time, then it also performs well in the number of nodes generated. Conversely, according to Figure 4, if significant reduction is achieved in the size of the state space, then the algorithm with abstraction also tends to perform well in terms of execution time (except for model bocdp). Moreover, as can be seen on both charts, within a group of models (c, m and n), the relative performance of the abstraction method tends to increase with increasing model complexity.

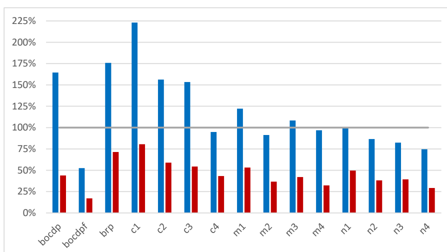

Fig. 3. Relative execution time and number of nodes generated of fastest configurations

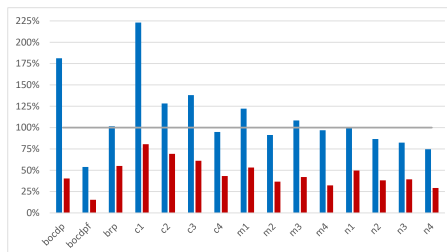

Fig. 4. Relative execution time and number of nodes generated of configurations with the smallest ASG

Moreover, for the models considered, configuration AID (Abstraction of discrete variables, Interpolation-based abstraction of clock variables, Depth-first search order) approximates the best configuration well for both execution time and ASG size, as this configuration tends to have a good performance on the more complex models. This is depicted on Figure 5 and Figure 6, where we compared configuration AID with the E-configurations in terms of execution time and size of the generated state space, respectively. In Figure 5, we denote by BEST the virtual best configuration, calculated from the best results of all other configurations. This data is omitted in Figure 6, as BEST greatly overlaps with configuration AID in terms of states generated. Moreover, to focus on the significant differences, we only depicted data for the hardest six models (denoted as $10 \dots 15$ on the horizontal axis) for each configuration.

5 Conclusions

In this paper we proposed a lazy algorithm for the location reachability problem of timed automata with discrete variables. The method is based on controlling

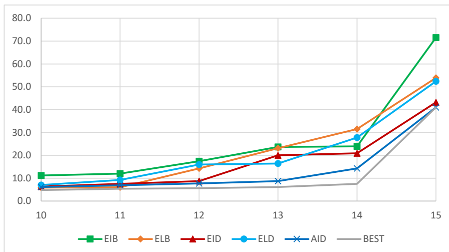

Fig. 5. Time to solve the hardest model instances (seconds)

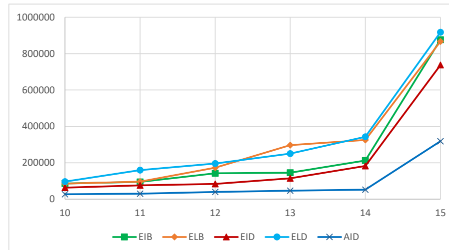

Fig. 6. Number of nodes generated for the hardest model instances

the visibility of discrete variables by using interpolation for valuations of variables. We demonstrated with experiments that our abstraction and refinement strategy, combined with lazy methods for the abstraction of continuous clock variables, can achieve significant reduction in the size of the generated state space during search, typically with low or no overhead in execution time, and in cases even with an additional speedup.

Future work. According to the method described in this paper, refinement is triggered upon encountering a disabled transition. In the future, we intend to experiment with counterexample-guided refinement for both the abstraction of discrete and continuous variables. In addition, we plan to experiment with different abstract domains (e.g. intervals), and investigate alternative refinement strategies for the discrete variables of timed systems. In particular we are interested in the performance for timed automata of the forward interpolation technique described in [4]. Moreover, we plan to explore more sophisticated strategies for finding covering states, as this can potentially yield considerable speedups for our method. Furthermore, although we evaluated our abstraction method in the context of timed systems, the technique itself can be applied in a more general context, and we plan to investigate its uses for model checking imperative programs.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer (2003). https://doi.org/10.1007/3-540-36577-X_18
3. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: TACAS 2004. LNCS, vol. 2988, pp. 312–326. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_25
4. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

5. Carioni, A., Ghilardi, S., Ranise, S.: Mcmt in the land of parametrized timed automata. In: 6th International Verification Workshop (VERIFY-2010). pp. 47–64 (2010)
6. 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). https://doi.org/10.1145/876638.876643
7. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer (1998). https://doi.org/10.1007/BFb0054180
8. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: FORMATS 2007. LNCS, vol. 4763, pp. 114–129. Springer (2007). https://doi.org/10.1007/978-3-540-75454-1_10
9. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages. pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
10. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. In: CAV 2013. LNCS, vol. 8044, pp. 990–1005. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_71
11. 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). https://doi.org/10.4230/LIPIcs.FSTTCS.2011.78
12. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. In: Logic in Computer Science. pp. 375–384. IEEE (2012). https://doi.org/10.1109/LICS.2012.48
13. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: Horn Clauses for Verification and Synthesis. EPTCS, vol. 169, pp. 39–52. Open Publishing Association (2014). https://doi.org/10.4204/EPTCS.169.6
14. Isenberg, T., Wehrheim, H.: Timed automata verification via IC3 with zones. In: ICFEM 2014. LNCS, vol. 8829, pp. 203–218. Springer (2014). https://doi.org/10.1007/978-3-319-11737-9_14
15. Kindermann, R., Junttila, T., Niemelä, I.: SMT-based induction methods for timed systems. In: FORMATS 2012. LNCS, vol. 7595, pp. 171–187. Springer (2012). https://doi.org/10.1007/978-3-642-33365-1_13
16. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. In: TACAS 2004. LNCS, vol. 2988, pp. 296–311. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_24
17. Morbé, G., Pigorsch, F., Scholl, C.: Fully symbolic model checking for timed automata. In: CAV 2011. LNCS, vol. 6806, pp. 616–632. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_50
18. Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., Majzik, I.: Theta: A framework for abstraction refinement-based model checking. In: Formal Methods in Computer Aided Design. pp. 176–179. FMCAD Inc. (2017). https://doi.org/10.23919/FMCAD.2017.8102257
19. Tóth, T., Majzik, I.: Lazy reachability checking for timed automata using interpolants. In: FORMATS 2017. LNCS, vol. 10419, pp. 264–280. Springer (2017). https://doi.org/10.1007/978-3-319-65765-3_15
20. Wang, W., Jiao, L.: Difference bound constraint abstraction for timed automata reachability checking. In: FORTE 2015. LNCS, vol. 9039, pp. 146–160. Springer (2015). https://doi.org/10.1007/978-3-319-19195-9_10