Acta Informatica (2022) 59:1–35
https://doi.org/10.1007/s00236-020-00393-4

ORIGINAL ARTICLE

Configurable verification of timed automata with discrete variables

Tamás Tóth[1] · **István Majzik**[1]

Received: 5 March 2019 / Accepted: 27 October 2020 / Published online: 15 December 2020

Abstract
Algorithms and protocols with time dependent behavior are often specified formally using timed automata. For practical real-time systems, besides real-valued clock variables, these specifications typically contain discrete data variables with nontrivial data flow. In this paper, we propose a configurable lazy abstraction framework for the location reachability problem of timed automata that potentially contain discrete variables. Moreover, based on our previous work, we uniformly formalize in our framework several abstraction refinement strategies for both clock and discrete variables that can be freely combined, resulting in many distinct algorithm configurations. Besides the proposed refinement strategies, the configurability of the framework allows the integration of existing efficient lazy abstraction algorithms for clock variables based on *LU*-bounds. We demonstrate the applicability of the framework and the proposed refinement strategies by an empirical evaluation on a wide range of timed automata models, including ones that contain discrete variables or diagonal constraints.

1 Introduction

Timed automata [1] is a widely used formalism for the modeling and verification of algorithms and protocols 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, or network of automata.

As timed automata contain real-valued clock variables, to ensure better performance and termination, model checkers for timed automata apply abstraction over clock variables. The

T. Tóth: This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary). I. Majzik: The research reported in this paper was partially supported by the BME NC TKP2020 grant of NKFIH Hungary.

✉ Tamás Tóth
totht@mit.bme.hu

István Majzik
majzik@mit.bme.hu

[1] Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

standard solution involves performing a forward exploration in the zone abstract domain [15], combined with extrapolation [4] parametrized by bounds appearing in guards in the model, extracted by static analysis [3]. Other zone-based methods propagate bounds lazily for all transitions [20] or along an infeasible path [22], and perform efficient inclusion checking with respect to a non-convex abstraction induced by the bounds [21]. Alternatively, some methods perform lazy abstraction directly over the zone abstract domain [30,33]. 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 (satisfiability modulo theories) [2] 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. Overall, we propose a *formal algorithmic framework* that enables the uniform formalization of several abstract domains and refinement strategies for both clock and discrete variables. The main elements are a generic algorithm for lazy reachability checking and an abstract reachability tree as its central data structure. Second, we present the *formalization and integration of several novel strategies* in this framework, especially based on our work on lazy interpolation over zones [30], and on the application of interpolation to lazily control the visibility of discrete variables in timed automata models [31]. The main advantage of the framework is that, based on the notion of the direct product abstract domain, it allows the *seamless combination* of various lazy abstraction methods, resulting in many distinct algorithm configurations that together admit efficient verification of a wide range of timed automata models. This algorithmic framework allowed a straightforward implementation of these strategies in our open source model checking framework THETA [29], this way enabling the practical *evaluation of the proposed algorithm configurations*. The configurability of this framework also allowed the integration of existing efficient lazy abstraction algorithms for clock variables based on *LU*-bounds [22], thus admitting the combination and comparison of our methods with the state-of-the-art.

We evaluated the algorithm configurations that our framework currently supports on a wide range of timed automata models, including ones that contain discrete variables or diagonal constraints. Our results show that our framework offers algorithm configurations that are competitive in terms of performance with the state-of-the-art.

1.1 Comparison to related work

Lazy abstraction [19], a form of counterexample-guided abstraction refinement [14], is an approach widely used for reachability checking, and in particular for model checking software. It consists of building an abstract reachability tree 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 [4] and on-the-fly propagation of bounds has been proposed [22]. A significant difference of this algorithm compared to usual lazy abstraction algorithms is that it builds an abstract reachability tree 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 that is disabled in the concrete system. Similar abstraction techniques based on building an ASG include difference bound constraint abstraction [33], the zone interpolation-based technique of [30], and the lazy abstraction method for timed automata with discrete variables proposed in [31]. In our work, we build on the same approach, but with

abstracting over the specifics of any given abstract domain, thus admitting configurability of the abstraction refinement strategy.

Symbolic handling of integer variables for timed automata is often supported by unbounded fully symbolic SMT-based approaches. Symbolic backward search techniques like [12,27] are based on the computation and satisfiability checking of pre-images. In [23], reachability checking for timed automata is addressed by solving Horn clauses. In the IC3-based [11] technique of [25], the problem of discrete variables is not addressed directly, but the possibility of generalization over discrete variables is (to some extent) inherent in the technique. In [24], also based on IC3, generalization of counterexamples to induction is addressed for both discrete and clock variables by zone-based pre-image computation. The abstraction methods proposed in our work are *completely theory agnostic*, and do not rely on an SMT-solver.

In [16], 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 [7]. 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 both forward and backward*, using post-image and weakest precondition computation, respectively. 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.

Timed automata with diagonal constraints are exponentially more concise than diagonal-free timed automata [9]. In [6], a method has been proposed that eliminates diagonal constraints occurring in timed automata specifications, resulting in an (in general) exponential blowup in the size of the automaton. An extrapolation method has been proposed in [5] that handles diagonal constraints on-the-fly. A refinement-based approach has been described in [10] that does not remove all diagonal constraints systematically. Instead, it performs forward model checking using the standard extrapolation operator used for diagonal-free timed automata, which might admit false negatives. In case a counterexample is found, it is analyzed for feasibility. If the counterexample is spurious, a set of diagonal constraints is selected and eliminated from the model, resulting in a new model, which is then fed back to the model checker. An implementation of the algorithm is described in [28]. In [18], the *LU*-abstraction based simulation relation of [4] is extended to models with diagonal constraints. The corresponding simulation test, which generalizes the inclusion test defined in [21] for the diagonal-free setting, is shown to be NP-complete, and is implemented in terms of SMT solving. In our work, we examine two methods for analyzing timed automata with diagonal

constraints. The first is based on the eager elimination of diagonal constraints, however, as our algorithms support discrete variables, instead of introducing new locations, we *introduce a new discrete variable per constraint*. In case abstraction refinement is used for these variables [31], a method is obtained that considers constraints as needed, similarly to [10]. However, instead of building a new model and running the model checker from scratch, this method is lazy, and performs abstraction refinement *locally in the state space where more precision is necessary*. The second approach is based on zone interpolation, which supports diagonal constraints, as well as other extensions [30], automatically. Thus in this case, elimination of diagonal constraints is not necessary. Unfortunately, this method is not complete, as it does not guarantee termination on all models.

This paper is based on our previous work presented in [30,31]. In addition to the results presented there, we provide an algorithmic framework in which we uniformly formalize, prove correct and evaluate our abstraction refinement strategies and their combinations. Moreover, besides the refinement strategy presented in [31] that propagates interpolants backward, we introduce a novel strategy that performs abstraction refinement by forward propagation of interpolants. Furthermore, we present an empirical evaluation of the algorithm configurations that the framework offers on a benchmark containing 51 timed automata models. In particular, we examine how the different configurations perform on models containing diagonal constraints.

1.2 Organization of the paper

The rest of the paper is organized as follows. In Sect. 2, we define the notations used throughout the paper, and present the theoretical background of our work. In Sect. 3 we propose our formal framework, a uniform lazy reachability checking algorithm that admits various abstract domains and refinement strategies. In Sect. 4, four abstraction refinement strategies are formalized in our framework, two for clock variables, and two for the efficient handling of discrete variables. Section 5 describes experiments performed on the proposed algorithm configurations. Finally, conclusions are given in Sect. 6.

2 Background and notations

In this section, we summarize the theoretical background of our work. Moreover, we define the notation used throughout the paper.

2.1 Valuations

Let C be a set of *clock variables* over $\mathbb{R}_{\geq 0}$, and D a set of *data variables* over $\mathbb{Z}$. Let $V = C \cup D$ denote the set of all variables.

A *clock constraint* is a formula $\varphi \in Constr_C$ that is a conjunction of atoms of the form $c \bowtie m$ and $c_i - c_j \bowtie m$ where $c, c_i, c_j \in C$ and $m \in \mathbb{Z}$ and $\bowtie \in \{<, \leq, >, \geq, \doteq\}$. In the latter case, if $i \neq j$, then a constraint is called a *diagonal constraint*. A *data constraint* is a well-formed formula $\varphi \in Constr_D$ built from variables in D and arbitrary function and predicate symbols interpreted over $\mathbb{Z}$. Let $Constr = Constr_C \cup Constr_D$ denote the set of all constraints.

A *clock update* (clock reset) is an assignment $u \in Update_C$ of the form $c := m$ where $c \in C$ and $m \in \mathbb{Z}$. A *data update* is an assignment $u \in Update_D$ of the form $d := t$ where

$d \in D$ and t is a term built from variables in D and function symbols interpreted over $\mathbb{Z}$. Let $Update = Update_C \cup Update_D$ denote the set of all updates.

The set of variables appearing in a term t (resp. in a formula φ) is denoted by $\mathsf{vars}(t)$ (resp. by $\mathsf{vars}(\varphi)$). Similarly, the set of variables occurring in an update is denoted by $\mathsf{vars}(u)$, that is, $\mathsf{vars}(x := t) = \mathsf{vars}(t) \cup \{x\}$.

A *valuation* over a finite set of variables is a function that maps variables to their respective domains. We will denote by $\mathcal{V}(X)$ the set of valuations over a set of variables X. Throughout the paper we will allow partial functions as valuations. We will denote by $\mathsf{def}(\sigma)$ the domain of definition of a valuation σ, that is, $\mathsf{def}(\sigma) = \{x \mid \sigma(x) \neq \bot\}$. 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. Let $[\![\varphi]\!]$ stand for the set of models of a formula φ, formally defined as $[\![\varphi]\!] = \{\sigma \in (\mathcal{V} \circ \mathsf{vars})(\varphi) \mid \sigma \models \varphi\}$, where $\circ$ denotes function composition as usual. Given a valuation σ, we denote by $\mathsf{form}(\sigma)$ the formula characterizing the valuation, that is, $\mathsf{form}(\sigma) = \bigwedge_{x \in \mathsf{def}(\sigma)} x \doteq \sigma(x)$.

Remark 1 Note that in the context of partial valuations, $\sigma \models \neg\varphi$ is a strictly stronger statement than $\sigma \not\models \varphi$. For example, $\{x \hookleftarrow 1\} \not\models y \doteq 1$ but it is not the case that $\{x \hookleftarrow 1\} \models y \not\doteq 1$.

Let $\sigma \preceq \sigma'$ iff $\sigma(x) = \sigma'(x)$ for all $x \in \mathsf{def}(\sigma')$. Moreover, let $A \preceq B$ iff for all $\sigma \in A$ there exists $\sigma' \in B$ such that $\sigma \preceq \sigma'$. Clearly, $\preceq$ is a partial order over sets of valuations. We will denote the restriction of valuation σ to a set of variables X by $\sigma\restriction_X$, that is, $(\sigma\restriction_X)(x) = \sigma(x)$ if $x \in X$ and $(\sigma\restriction_X)(x) = \bot$ if $x \notin X$. We lift the notion to sets of valuations with the obvious meaning. Let moreover $(\!|\sigma|\!) = \{\sigma' \in \mathcal{V}(V) \mid \sigma' \preceq \sigma\}$, also defined for sets of valuations in the obvious way.

We state the following lemmas (without proof).

Lemma 1 $\sigma \preceq \sigma' \Rightarrow \sigma' \models \varphi \Rightarrow \sigma \models \varphi$

Lemma 2 $\sigma \preceq \sigma' \Leftrightarrow \sigma \models \mathit{form}(\sigma')$

Lemma 3 $A \preceq B \Rightarrow A\restriction_X \preceq B\restriction_X$

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

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

if $\sigma(x) = \sigma'(x)$ for all $x \in \mathsf{def}(\sigma) \cap \mathsf{def}(\sigma')$, and is undefined otherwise. We extend this function to sets of valuations in both parameters in the obvious way.

Finally, given a valuation σ and an update $x := t$, we denote by $\sigma\{x := t\}$ the valuation σ' such that $\sigma'(x) = \sigma(t)$ and $\sigma'(x') = \sigma(x')$ for all $x' \neq x$. For a sequence of updates, let $\sigma\{\epsilon\} = \sigma$ and $\sigma\{u \cdot \mu\} = \sigma\{u\}\{\mu\}$, where u is an update and μ is a sequence of updates.

2.2 Timed automata

In the area of modeling and verifying time-dependent behavior, 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 $\mathcal{A} = (L, C, D, T, \ell_0)$ where

- L is a finite set of locations,
- C is a finite set of continuous clock variables over $\mathbb{R}_{\geq 0}$,
- D is a finite set of discrete data variables over $\mathbb{Z}$,
- $T \subseteq L \times \mathcal{P}(Constr) \times Update^* \times L$ is a finite set of transitions, where for a transition (ℓ, G, μ, ℓ'), the set $G \subseteq Constr$ is a set of guards, and $\mu \in Update^*$ is a sequence of updates, and
- $\ell_0 \in L$ is the initial location.

We are going to refer to a sequence of transitions $\pi \in T^*$ as a *path*.

Remark 2 According to the above definition, clearly $C \cap D = \emptyset$. Note that given a guard $g \in G$, either $\mathsf{vars}(g) \subseteq C$, or $\mathsf{vars}(g) \subseteq D$. Similarly, given an update u, either $\mathsf{vars}(u) \subseteq C$, or $\mathsf{vars}(u) \subseteq D$.

Definition 2 (*Semantics*) Let σ_0 be the unique total function $\sigma_0 : V \to \{0\}$. The operational semantics of a timed automaton is given by a labeled transition system with initial state (ℓ_0, σ_0) and two kinds of transitions:

- *Delay:* $(\ell, \sigma) \xrightarrow{\delta} (\ell', \sigma')$ for some real number $\delta \geq 0$ where $\ell' = \ell$ and $\sigma' = delay_\delta(\sigma)$ with

$$delay_\delta(\sigma)(x) = \begin{cases} \sigma(x) + \delta & \text{if } x \in C \\ \sigma(x) & \text{otherwise} \end{cases}$$

- *Action:* $(\ell, \sigma) \xrightarrow{t} (\ell', \sigma')$ for some transition $t = (\ell, G, \mu, \ell')$ where $\sigma' = action_t(\sigma)$ with

$$action_t(\sigma) = \begin{cases} \bot & \text{if } \sigma \models \neg g \text{ for some } g \in G \\ \sigma\{\mu\} & \text{otherwise} \end{cases}$$

We will use the notation $\mathcal{C} = \mathcal{V}(V)$, and refer to a valuation $\sigma \in \mathcal{C}$ as a *concrete state*. A *state* of a timed automaton is a state of its semantics, that is, a pair (ℓ, σ) where $\ell \in L$ and $\sigma \in \mathcal{C}$.

In case $D = \emptyset$, the above definition for semantics coincides with the semantics of timed automata in the usual sense. Throughout the paper, we will refer to a timed automaton with discrete variables simply as a timed automaton.

Definition 3 (*Run*) A *run* of a timed automaton is a sequence of states from the initial state (ℓ_0, σ_0) along the transition relation

$$(\ell_0, \sigma_0) \xrightarrow{\alpha_1} (\ell_1, \sigma_1) \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} (\ell_k, \sigma_k)$$

where $\alpha_i \in T \cup \mathbb{R}_{\geq 0}$ for all $0 \leq i \leq k$.

Definition 4 (*Reachable location*) A location $\ell \in L$ is *reachable* iff there exists a run such that $\ell_k = \ell$.

Clearly, if a location is reachable then it is reachable along a run of the form $\cdot \xrightarrow{\delta_0} (\ell_0, \sigma_0') \xrightarrow{t_1} \cdot \xrightarrow{\delta_1} (\ell_1, \sigma_1) \xrightarrow{t_2} \cdot \xrightarrow{\delta_2} \cdots \xrightarrow{t_k} \cdot \xrightarrow{\delta_k} (\ell_k, \sigma_k)$. This observation enables the definition of a symbolic semantics for timed automata as follows.

Definition 5 (*Symbolic semantics*) Let $\Sigma_0 = \{delay_\delta(\sigma_0) \mid \delta \geq 0\}$, that is, the set of concrete states reachable from σ_0 by a delay transition. The symbolic semantics of a timed automaton is a labeled transition system with initial state (ℓ_0, Σ_0) and transitions of the form $(\ell, \Sigma) \xrightarrow{t} (\ell', \Sigma')$ where $t = (\ell, \cdot, \cdot, \ell')$ and $\Sigma' = post_t(\Sigma)$ with the concrete post-image operator

$$post_t(\sigma) = \{(delay_\delta \circ action_t)(\sigma) \mid \delta \geq 0\},$$

defined for paths as $post_\epsilon = id$ and $post_{t \cdot \pi} = post_\pi \circ post_t$.

We will refer to a pair (ℓ, Σ) with $\ell \in L$ and $\Sigma \subseteq \mathcal{C}$ as a *symbolic state*.

Let $pre_t = post_t^{-1}$ and $post_t^X(\Sigma) = post_t(\Sigma)\uparrow_X$ for $X \in \{C, D\}$. Let moreover $pre_t^C = (post_t^C)^{-1}$. Furthermore, let $\nu_0 = \Sigma_0\uparrow_D$ and $Z_0 = \Sigma_0\uparrow_C$.

Remark 3 As a consequence of Remark 2, it can be shown that in general, a symbolic state (ℓ, Σ) occurring in a symbolic run of timed automaton is such that $\Sigma = \nu \otimes Z$, where $\nu = \Sigma\uparrow_D$ is a data valuation, and $Z = \Sigma\uparrow_C$ is a special set of clock valuations, called a zone (see Sect. 2.3). Moreover, $post_t(\nu \otimes Z) = post_t^D(\nu) \otimes post_t^C(Z)$.

Definition 6 (*Enabled transition*) Given a set of concrete states Σ, we will say that a transition t is *enabled* from Σ iff $post_t(\Sigma) \neq \emptyset$, otherwise it is *disabled*.

Definition 7 (*Feasible path*) We will say that a path π is *feasible* (resp. *data-feasible*) (resp. *clock-feasible*) iff $post_\pi(\Sigma_0) \neq \emptyset$ (resp. $post_\pi^D(\nu_0) \neq \emptyset$) (resp. $post_\pi^C(Z_0) \neq \emptyset$), otherwise it is *infeasible* (resp. *data-infeasible*) (resp. *clock-infeasible*).

Remark 4 By Remark 3 and induction, a path π is feasible iff it is data-feasible and clock-feasible.

Definition 8 (*Symbolic run*) A *symbolic run* of a timed automaton is a sequence of symbolic states from the symbolic initial state (ℓ_0, Σ_0) along the transition relation $(\ell_0, \Sigma_0) \xrightarrow{t_1} (\ell_1, \Sigma_1) \xrightarrow{t_2} \cdots \xrightarrow{t_k} (\ell_k, \Sigma_k)$ where $\Sigma_k \neq \emptyset$.

Remark 5 Clearly, a timed automaton has a symbolic run $\cdot \xrightarrow{t_1} \cdot \xrightarrow{t_2} \cdots \xrightarrow{t_k} \cdot$ iff the path $t_1 t_2 \cdots t_k$ is feasible.

Proposition 1 *For a timed automaton, a location $\ell \in L$ is reachable iff there exists a symbolic run with $\ell_k = \ell$* [15].

2.3 Zones and DBMs

A *zone* $Z \in \mathcal{Z}$ is the set of solutions of a clock constraint $\varphi \in Constr_C$, that is $Z = \{\eta \in \mathcal{V}(C) \mid \eta \models \varphi\}$. If Z and Z' are zones and $t \in T$, then $\emptyset$, $\mathcal{V}(C)$, Z_0, $Z \cap Z'$, $post_t^C(Z)$ and $pre_t^C(Z')$ are also zones. In the context of zones, we will denote $\emptyset$ by $\bot$ and

$\mathcal{V}(C)$ by $\top$. Zones are not closed under complementation, but the complement of any zone is the union of finitely many zones. For a zone Z, we are going to denote a minimal set of such zones by $\neg Z$.

Zones can be efficiently represented by difference bound matrices [17]. A *bound* is either ∞, or a finite bound of the form $(m, \prec)$ where $m \in \mathbb{Z}$ and $\prec \in \{<, \leq\}$. Difference bounds can be totally ordered by "strength", that is, $(m, \prec) < \infty$ and $(m_1, \prec_1) < (m_2, \prec_2)$ iff $m_1 < m_2$ and $(m, <) < (m, \leq)$. Moreover the sum of two bounds is defined as $b + \infty = \infty$ and $(m_1, \leq) + (m_2, \leq) = (m_1 + m_2, \leq)$ and $(m_1, <) + (m_2, \prec) = (m_1 + m_2, <)$.

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

Proposition 2 *A DBM M is inconsistent iff there exists a negative cycle in M, that is, a set of pairs of indexes $\{(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)\}$ such that $M_{i_1,i_2} + \cdots + M_{i_{k-1},i_k} + M_{i_k,i_1} < (0, \leq)$* [17].

For a consistent DBM M, we say it is *canonical* iff constraints in it cannot be strengthened without losing solutions, formally, iff $M_{i,i} = (0, \leq)$ for all $0 \leq i \leq n$ and $M_{i,j} \leq M_{i,k} + M_{k,j}$ for all $0 \leq i, j, k \leq n$. For convenience, we will also consider the inconsistent DBM M with the single finite bound $M_{0,0} = (0, <)$ canonical. Up to the ordering of clocks, the canonical form is unique.

The zone operations described above, as well as set inclusion $\subseteq$ over zones, can be efficiently implemented in terms of canonical DBMs [5]. Therefore, we will refer to a canonical DBM M (syntax) and the zone $[\![M]\!]$ it represents (semantics) interchangeably throughout the paper.

Moreover, for two DBMs M_1 and M_2, we will denote by $\min(M_1, M_2)$ the (not necessarily canonical) DBM M where $M_{i,j} = \min(M_{1,ij}, M_{2,ij})$. It can be easily shown that $[\![\min(M_1, M_2)]\!] = [\![M_1]\!] \cap [\![M_2]\!]$.

3 Algorithm for lazy reachability checking

In this section we present our uniform approach, a lazy reachability checking algorithm that allows the combination of various abstract domains and refinement strategies. It is based on the notion of Abstract Reachability Tree, which is defined in the sequel. Then the algorithm itself is described.

3.1 Abstract reachability tree

The central data structure of the algorithm is an abstract reachability tree.

Definition 9 (*Abstract domain*) For our purposes, an abstract domain for a timed automaton is a tuple $\mathbb{D} = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ such that

- $\mathcal{S}$ is set of abstract states,
- $\sqsubseteq \subseteq \mathcal{S} \times \mathcal{S}$ is a preorder,
- $\mathsf{init} \in \mathcal{S}$ is the abstract initial state,
- $\mathsf{post} : T \times \mathcal{S} \to \mathcal{S}$ is the abstract post-image operator, and

– $[\![\cdot]\!] : \mathcal{S} \to \mathcal{P}(\mathcal{C})$ is the concretization function.

For soundness, we assume the following properties to hold.

Definition 10 (*Sound abstraction*) An abstract domain $(\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ is sound iff

– $s_1 \sqsubseteq s_2 \Rightarrow [\![s_1]\!] \subseteq [\![s_2]\!]$,
– $\Sigma_0 \subseteq [\![\mathsf{init}]\!]$, and
– $post_t[\![s]\!] \subseteq [\![\mathsf{post}_t(s)]\!]$.

Definition 11 (*Unwinding*) An unwinding of a timed automaton $\mathcal{A}$ is a tuple $U = (N, E, n_0, M_N, M_E, \triangleright)$ 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
– $\triangleright \subseteq N \times N$ is the covering relation.

For an unwinding we require that the following properties hold:

– $M_N(n_0) = \ell_0$,
– for each edge $e \in E$ with $e = (n, n')$ the transition $M_E(e) = (\ell, \cdot, \cdot, \ell')$ is such that $M_N(n) = \ell$ and $M_N(n') = \ell'$,
– for all nodes n and n' such that $n \triangleright n'$ it holds that $M_N(n) = M_N(n')$.

The term $n \triangleright n'$ marks that search from node n of the unwinding is to be pruned, as another node n' admits all runs that are feasible from n. We define the following shorthand notations for convenience: $\ell_n = M_N(n)$ and $t_e = M_E(e)$.

Definition 12 (*Abstract reachability tree*) An abstract reachability tree (ART) for a timed automaton $\mathcal{A}$ over a sound abstract domain $\mathbb{D}$ is a labeled unwinding, that is, a pair $\mathcal{G} = (U, \psi)$ where

– U is an unwinding of $\mathcal{A}$, and
– $\psi : N \to \mathcal{S}$ is a labeling of nodes by abstract states.

We will use the following shorthand notation: $s_n = \psi(n)$.

Definition 13 (*Properties of nodes*) 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 $[\![s_n]\!]$, or n has a successor for t. A node n is *covered* iff $n \triangleright n'$ 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 ART to be useful for reachability checking, we have to ensure that the tree represents an over-approximation of the set of reachable states. Therefore we introduce restrictions on the labeling, as formalized in the next definition.

Definition 14 (*Well-labeled node*) A node n of an ART $\mathcal{G}$ for a timed automaton $\mathcal{A}$ is well-labeled iff the following conditions hold:

– (initiation) if $n = n_0$, then $\Sigma_0 \subseteq [\![s_n]\!]$,
– (consecution) if $n \neq n_0$, then for its parent m and the transition $t = t_{(m,n)}$ it holds that $post_t[\![s_m]\!] \subseteq [\![s_n]\!]$
– (coverage) if $n \triangleright n'$ for some node n', then $[\![s_n]\!] \subseteq [\![s_{n'}]\!]$ and n' is not excluded.

Besides preserving reachable states, we will also ensure that nodes represent runs of the automaton. We formalize this in the following definitions.

Definition 15 (*Feasible node and transition*) Let n be a node of an ART $\mathcal{G}$, and π the path from n_0 to n in $\mathcal{G}$. Then n is feasible iff π is feasible. Moreover, a transition t is feasible from n iff the path $\pi \cdot t$ is feasible.

The above definitions for nodes can be extended to trees.

Definition 16 (*Properties of ARTs*) An ART is complete, ℓ-safe, well-labeled or feasible iff all its nodes are complete, ℓ-safe, well-labeled, or feasible, respectively.

A well-labeled ART preserves reachable states, which is expressed by the following proposition.

Proposition 3 *Let $\mathcal{G}$ be a complete, well-labeled ART for a timed automaton $\mathcal{A}$. If $\mathcal{A}$ has a symbolic run $(\ell_0, \Sigma_0) \xrightarrow{t_1} (\ell_1, \Sigma_1) \xrightarrow{t_2} \cdots \xrightarrow{t_k} (\ell_k, \Sigma_k)$ then $\mathcal{G}$ has a non-excluded node n such that $\ell_k = \ell_n$ and $\Sigma_k \subseteq [\![s_n]\!]$.*

Proof We prove the statement by induction on the length k of the symbolic run. If $k = 0$, then $\ell_0 = \ell_{n_0}$ and $\Sigma_0 \subseteq [\![s_{n_0}]\!]$ by condition *initiation*, thus n_0 is a suitable witness. 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 $\Sigma_{k-1} \subseteq [\![s_m]\!]$.

Clearly transition t_k is not disabled from $[\![s_m]\!]$, as then by the induction hypothesis it would also be disabled from Σ_{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*, we have $post_{t_k}[\![s_m]\!] \subseteq [\![s_n]\!]$. As $\Sigma_{k-1} \subseteq [\![s_m]\!]$, by the monotonicity of images in $\subseteq$, we obtain $\Sigma_k \subseteq [\![s_n]\!]$.

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 $[\![s_n]\!] \subseteq [\![s_{n'}]\!]$ and n' is not excluded, thus n' is a suitable witness. □

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 constructs an ART for $\mathcal{A}$ and continually maintains its well-labeledness and feasibility. Upon termination, it either witnesses reachability of ℓ_e by a feasible node n such that $\ell_n = \ell_e$, which by Definition 15 corresponds to a symbolic run of $\mathcal{A}$ to ℓ_e, or produces a complete, well-labeled, ℓ_e-safe ART that proves unreachability of ℓ_e by Proposition 3.

The main data structures of the algorithm are the ART $\mathcal{G}$ and sets *passed* and *waiting*. Set *passed* is used to store nodes that are expanded, and *waiting* stores nodes that are incomplete. The algorithm consists of two subprocedures, CLOSE and EXPAND. Procedure CLOSE attempts to cover a node n by some other node. It calls a procedure COVER that tries to force cover the node by adjusting its label so that it is subsumed by the label of some candidate node n'. Procedure EXPAND expands a node n by creating its successors. To avoid creating infeasible nodes, it calls a procedure DISABLE that checks feasibility of a given transition t, and adjusts the labeling of n so that if t is infeasible from n, then it also becomes disabled from $[\![s_n]\!]$. Both CLOSE and EXPAND potentially modify the labeling of some nodes as a side effect, but in a way

Algorithm 1 Reachability algorithm

```
1: ensure ρ = SAFE iff ℓ_e is unreachable for A
2: function EXPLORE(A, ℓ_e) returns ρ ∈ {SAFE, UNSAFE}
3:     let n_0 be a node with ℓ_{n_0} = ℓ_0 and s_{n_0} = init
4:     N ← {n_0}, E ← ∅, ▷ ← ∅
5:     let G be an ART for A over N, E and ▷
6:
7:     passed ← ∅, waiting ← {n_0}
8:     invariant G is well-labeled and feasible
9:     while n ∈ waiting for some n do
10:        waiting ← waiting \ {n}
11:        if ℓ_n = ℓ_e then
12:            return UNSAFE
13:        else
14:            CLOSE(n)
15:            if n is not covered then
16:                EXPAND(n)
17:    return SAFE

18: invariant G is well-labeled and feasible
19: procedure CLOSE(n)
20:    for all n' ∈ passed such that ℓ_n = ℓ_{n'} do
21:        COVER(n, n')
22:        if s_n ⊑ s_{n'} then
23:            ▷ ← ▷ ∪ {(n, n')}
24:            return

25: invariant G is well-labeled and feasible
26: ensure n is expanded
27: procedure EXPAND(n)
28:    for all t ∈ T such that t = (ℓ, ·, ·, ℓ') with ℓ = ℓ_n do
29:        if not DISABLE(n, t) then
30:            let s' = post_t(s_n)
31:            let n' be a new node with ℓ_{n'} = ℓ' and s_{n'} = s'
32:            let e = (n, n') be a new edge with t_e = t
33:            N ← N ∪ {n'}
34:            E ← E ∪ {e}
35:            waiting ← waiting ∪ {n'}
36:    passed ← passed ∪ {n}

37: invariant G is well-labeled and feasible
38: procedure COVER(n, n')

39: invariant G is well-labeled and feasible
40: ensure β iff t is disabled from ⟦s_n⟧
41: ensure ¬β iff t is feasible from n
42: function DISABLE(n, t) returns β
```

that maintains well-labeledness and feasibility of the ART. Naturally, the implementation of procedures COVER and DISABLE depends on the abstract domain, and are described in Sect. 4 in detail.

The algorithm consists of a single loop in line 9 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 the 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 procedure contracts listed in Algorithm 1. We focus on partial correctness, as termination depends on the particular abstract domain and refinement method used. We note that in general, termination can be easily ensured using the right extrapolation operator for clock variables [22,30,33].

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

Proof (sketch) Let $covered = \{n \in N \mid n \text{ is covered}\}$. It is easy to verify that the algorithm maintains the following invariants:

- $N = passed \cup waiting \cup covered$,
- *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 and feasible, partial correctness of the algorithm is then a direct consequence: At line 12 a node is encountered that is not ℓ_e-safe, thus by Definition 15 there is a symbolic run of $\mathcal{A}$ to ℓ_e; conversely, at line 17 the set *waiting* is empty, so $\mathcal{G}$ is complete and ℓ_e-safe, and as a consequence of Proposition 3 the location ℓ_e is indeed unreachable for $\mathcal{A}$.

What remains to show is that the algorithm maintains well-labeledness and feasibility of $\mathcal{G}$. We assume that procedures COVER and DISABLE maintain well-labeledness and feasibility, which we prove to hold in Sect. 4.

Initially, node n_0 is well-labeled, as $\Sigma_0 \subseteq [\![\mathsf{init}]\!] = [\![s_{n_0}]\!]$, thus n_0 satisfies *initiation*. It also trivially satisfies feasibility, as $post_\epsilon(\Sigma_0) = \Sigma_0 \neq \emptyset$. Procedure CLOSE trivially maintains well-labeledness and feasibility, as it just possibly adds a covering edge for two nodes such that condition *coverage* is not violated. In procedure EXPAND, if DISABLE(n, t) for a transition t, then t is not feasible from n, and the labeling is adjusted so that t is disabled from $[\![s_n]\!]$. Otherwise, t is feasible from n, and a successor node n' is created. Clearly, n' is feasible as t is feasible. Moreover, $post_t[\![s_n]\!] \subseteq [\![\mathsf{post}_t(s_n)]\!] = [\![s_{n'}]\!]$, thus n' satisfies *consecution*. Thus according to the contract, n becomes expanded, and all its successors are well-labeled and feasible, so well-labeledness and feasibility of $\mathcal{G}$ is preserved. □

4 Abstraction refinement

Algorithm 1 is abstracted over the particular abstract domain used to well-label the constructed ART. Moreover, it declares two procedures, COVER and DISABLE, that perform forced covering and abstraction refinement over the abstract domain, respectively. In this section, we describe several possible abstract domains, and corresponding abstraction refinement strategies, that can be used for model checking timed automata with discrete variables.

In the listings of the given refinement strategies, we are going to refer to a simple procedure UPDATE that enables safely updating the labeling for a given node in the ART.

Proposition 5 UPDATE *is totally correct: If either* n *is the root and* $\Sigma_0 \subseteq [\![s]\!]$, *or there exists an edge* $e = (m, n)$ *with* $t_e = t$ *for some* m *and* $post_t[\![s_m]\!] \subseteq [\![s]\!]$, *then* UPDATE($n$, s) *terminates and ensures* $s_n = s$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Algorithm 2 Safely updating the abstraction

1: **invariant** $\mathcal{G}$ is well-labeled and feasible
2: **require** n root $\Rightarrow$ $\Sigma_0 \subseteq [\![s]\!]$
3: **require** $(m, n) \in E$ with $t = t_{(m,n)}$ for some $m \Rightarrow post_t [\![s_m]\!] \subseteq [\![s]\!]$
4: **ensure** $s_n = s$
5: **procedure** UPDATE(n, s)
6: **for all** m such that $m \triangleright n$ and $s_m \not\sqsubseteq s$ **do**
7: $\triangleright \leftarrow \triangleright \setminus (m, n)$
8: $waiting \leftarrow waiting \cup \{m\}$
9: $s_n \leftarrow s$

Proof Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. At the end of the procedure, $s_n = s$ is ensured. Clearly, n is well-labeled: *initiation* and *consecution* is ensured by contract, and *coverage* is ensured by the loop due to soundness of the abstract domain. □

4.1 Combination of abstractions

Our approach is based on the direct product of abstract domains, as described below.

Definition 17 (*Direct product domain*) Let $\mathbb{D}_i = (\mathcal{S}_i, \sqsubseteq_i, \mathsf{init}_i, \mathsf{post}^i, [\![\cdot]\!]_i)$ for $i \in \{1, 2\}$. Then their direct product is the abstract domain $\mathbb{D}_1 \times \mathbb{D}_2 = (S, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ where

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$,
- $(s_1, s_2) \sqsubseteq (s_1', s_2')$ iff $s_1 \sqsubseteq_1 s_1'$ and $s_2 \sqsubseteq_2 s_2'$ (thus $\sqsubseteq$ is a preorder),
- $\mathsf{init} = (\mathsf{init}_1, \mathsf{init}_2)$,
- $\mathsf{post}_t(s_1, s_2) = (\mathsf{post}_t^1(s_1), \mathsf{post}_t^2(s_2))$, and
- $[\![(s_1, s_2)]\!] = [\![s_1]\!]_1 \cap [\![s_2]\!]_2$.

In later descriptions, when it is clear from the context, we are going to omit indexes when referring to components of a direct product (and write e.g. $(\mathsf{post}_t(s_1), \mathsf{post}_t(s_2))$ instead of $(\mathsf{post}_t^1(s_1), \mathsf{post}_t^2(s_2))$).

Proposition 6 *If $\mathbb{D}_1$ and $\mathbb{D}_2$ are sound, then $\mathbb{D}_1 \times \mathbb{D}_2$ is sound.*

In case of timed automata with discrete variables, as according to Definition 1, abstraction and refinement can be conveniently defined compositionally, where clock variables and discrete variables are handled by separate abstractions. Algorithm 3 describes a straightforward method for achieving this separation.

In the above description, in line 10 and line 15, we refer to the preservation of well labeledness for the two projections of the ART. This weaker assumption will simplify proofs of correctness for the component refiners. We show that this implies well-labeledness in the original sense.

Total correctness of $\text{COVER}_\times$ follows from total correctness of COVER_D and COVER_C. We show total correctness of $\text{DISABLE}_\times$ as follows.

Proposition 7 $\text{DISABLE}_\times$ *is totally correct:* $\text{DISABLE}_\times(n, t)$ *terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; Moreover, it returns* false *iff t is feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Algorithm 3 Combination of Abstractions

1: **procedure** $\text{COVER}_\times(n, n')$
2: $\quad \text{COVER}_D(n, n')$
3: $\quad \text{COVER}_C(n, n')$

4: **invariant** $\mathcal{G}$ is well-labeled and feasible
5: **procedure** $\text{COVER}_D(n, n')$

6: **invariant** $\mathcal{G}$ is well-labeled and feasible
7: **procedure** $\text{COVER}_C(n, n')$

8: **function** $\text{DISABLE}_\times(n, t)$ **returns** β
9: $\quad$ **return** $\text{DISABLE}_D(n, t)$ or $\text{DISABLE}_C(n, t)$

10: **invariant** $\mathcal{G}$ is well-labeled and feasible
11: **define** $(s_1, s_2) = s_n$
12: **ensure** β iff t is disabled from $[\![s_1]\!]$
13: **ensure** $\neg\beta$ iff t is data-feasible from n
14: **function** $\text{DISABLE}_D(n, t)$ **returns** β

15: **invariant** $\mathcal{G}$ is well-labeled and feasible
16: **define** $(s_1, s_2) = s_n$
17: **ensure** β iff t is disabled from $[\![s_2]\!]$
18: **ensure** $\neg\beta$ iff t is clock-feasible from n
19: **function** $\text{DISABLE}_C(n, t)$ **returns** β

Proof Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes.

First we show that $\text{DISABLE}_\times$ maintains well-labeledness. By contract, DISABLE_C and DISABLE_D preserve well-labeledness of $\mathcal{G}$ (in the weaker sense described above). Let $s_n = (s_1, s_2)$ for root node n. As $\Sigma_0 \subseteq [\![s_1]\!]$ and $\Sigma_0 \subseteq [\![s_2]\!]$, clearly $\Sigma_0 \subseteq [\![s_1]\!] \cap [\![s_2]\!] = [\![(s_1, s_2)]\!]$, thus *initiation* is preserved. Now let $s_m = (s_1, s_2)$ and $s_n = (s_1', s_2')$ for nodes m and n such that $(m, n) \in E$ and $t = t_{(m,n)}$. As $post_t$ is an image and $post_t[\![s_1]\!] \subseteq [\![s_1']\!]$ and $post_t[\![s_2]\!] \subseteq [\![s_2']\!]$, we have $post_t[\![(s_1, s_2)]\!] = post_t([\![s_1]\!] \cap [\![s_2]\!]) \subseteq post_t[\![s_1]\!] \cap post_t[\![s_2]\!] \subseteq [\![s_1']\!] \cap [\![s_2']\!] = [\![(s_1', s_2')]\!]$, thus *consecution* is preserved. Finally, let $s_m = (s_1, s_2)$ and $s_n = (s_1', s_2')$ for nodes m and n such that $m \triangleright n$. As $[\![s_1]\!] \subseteq [\![s_1']\!]$ and $[\![s_2]\!] \subseteq [\![s_2']\!]$, clearly $[\![(s_1, s_2)]\!] = [\![s_1]\!] \cap [\![s_2]\!] \subseteq [\![s_1']\!] \cap [\![s_2']\!] = [\![(s_1', s_2')]\!]$, thus *coverage* is preserved.

Assume that t is feasible from n. Then t is both data- and clock-feasible from n by Remark 4. Thus $\text{DISABLE}_D(n, t) = \mathsf{false}$ and $\text{DISABLE}_C(n, t) = \mathsf{false}$ by contract, from which $\text{DISABLE}_\times(n, t) = \mathsf{false}$ follows directly. Assume that t is not feasible from n. Then t is either not data- or not clock-feasible from n by Remark 4. Assume t is not data-feasible from n. Thus $\text{DISABLE}_D(n, t) = \mathsf{true}$ and t becomes disabled from $[\![s_1]\!]$ by contract. As a consequence, $\text{DISABLE}_\times(n, t) = \mathsf{true}$, and t becomes disabled from $[\![s_n]\!] = [\![(s_1, s_2)]\!] = [\![s_1]\!] \cap [\![s_2]\!]$. The other case follows symmetrically. □

To simplify exposition, we are going to treat ψ as a lens (in simple terms, a pair consisting of a "getter" and a "setter") that can be used to deeply manipulate the structure of a given label. Thus later in the text, when we refer to s_n, we are going to mean the corresponding component of a direct product based on the context.

4.2 Abstraction for clock variables

First, we address abstraction refinement over clock variables.

4.2.1 Zone abstraction

Most model checkers for timed automata rely on zones for abstracting clock valuations. We define zone abstraction in our framework as follows.

Definition 18 (*Zone abstraction*) We define zone abstraction as the abstract domain $\mathbb{D}_{\mathcal{Z}} = (\mathcal{Z}, \subseteq, Z_0, post^C, (\!|\cdot|\!))$.

Note that in the absence of discrete variables, Definition 18 corresponds to the usual definition of zone abstraction.

Proposition 8 *$\mathbb{D}_{\mathcal{Z}}$ is sound.*

We define $\text{COVER}_{\mathcal{Z}}$ as a no-op, thus its total correctness is trivial. Moreover, we define $\text{DISABLE}_{\mathcal{Z}}$ as $\text{DISABLE}_{\mathcal{Z}}(n, t)$ iff $\mathsf{post}_t(Z) \sqsubseteq \bot$ for $Z = s_n$.

Proposition 9 $\text{DISABLE}_{\mathcal{Z}}$ *is totally correct:* $\text{DISABLE}_{\mathcal{Z}}(n, t)$ *terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns* false *iff t is clock-feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof Termination of the procedure is trivial. Well-labeledness and feasibility follow from the fact that the procedure has no side effects. Let π be the path induced by n. Notice that $Z = post^C_\pi(Z_0)$. Assume $post^C_t(Z) \neq \bot$. Then by definition, t is clock-feasible from n, and the procedure returns false. Now assume $post^C_t(Z) = \bot$. Then by definition, t is not clock-feasible from n. But t is also disabled from $(\!|Z|\!)$, and the procedure returns true. □

4.2.2 Lazy zone abstraction

To obtain a coarser abstraction, we extend zone abstraction with interpolation as follows.

Definition 19 (*Lazy zone abstraction*) Let $\mathbb{D}_{\mathcal{ZI}} = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ be the abstract domain over $\mathbb{D}_{\mathcal{Z}}$ with

- $\mathcal{S} = \mathcal{Z} \times \mathcal{Z}$,
- $(Z, W) \sqsubseteq (Z', W')$ iff $W \sqsubseteq W'$,
- $\mathsf{init} = (\mathsf{init}, \top)$,
- $\mathsf{post}_t(Z, W) = (\mathsf{post}_t(Z), \top)$, and
- $[\![(Z, W)]\!] = [\![W]\!]$.

Proposition 10 *$\mathbb{D}_{\mathcal{ZI}}$ is sound.*

Given an abstract state (Z, W), the purpose of Z is to encode an exact set of reachable valuations, whereas the purpose of W is to represent a safe overapproximation of Z. This potentially enables better coverage between nodes, thus faster convergence, compared to the purely zone-based setting. In order to efficiently maintain this relationship however, we have to define procedure $\text{COVER}_{\mathcal{ZI}}$ and $\text{DISABLE}_{\mathcal{ZI}}$ accordingly. To maintain well-labeledness, these procedures rely on a procedure BLOCK that performs abstraction refinement by safely adjusting labels of nodes.

In $\text{COVER}_{\mathcal{ZI}}$, as $Z \subseteq W'$ and $B \cap W' \subseteq \bot$, clearly $Z \cap B \subseteq \bot$, thus calling $\text{BLOCK}(n, B)$ is safe. Other than that, total correctness of $\text{COVER}_{\mathcal{ZI}}$ follows trivially from total correctness of BLOCK (see later). To show the correctness of $\text{DISABLE}_{\mathcal{ZI}}$, we state the following simple lemma that establishes a connection between pre^C and $post^C$.

Algorithm 4 Lazy Zone Abstraction

```
1: procedure COVER_{ZI}(n, n')
2:     let (Z, ·) = s_n
3:     let (·, W') = s_{n'}
4:     if Z ⊆ W' then
5:         for all B ∈ ¬W' do
6:             BLOCK(n, B)
7: function DISABLE_{ZI}(n, t)
8:     let (Z, W) = s_n
9:     let Z' = post_t^C(Z)
10:    if Z' = ⊥ then
11:        BLOCK(n, pre_t^C(⊤))
12:        return true
13:    else
14:        return false

15: invariant G is well-labeled and feasible
16: define (Z, W) = s_n
17: require Z ∩ B ⊆ ⊥
18: ensure W ∩ B ⊆ ⊥
19: procedure BLOCK(n, B)
```

Lemma 4 $Z \cap pre_t^C(Z') \subseteq \bot \Leftrightarrow post_t^C(Z) \cap Z' \subseteq \bot$

Proposition 11 DISABLE$_{\mathcal{ZI}}$ *is totally correct:* DISABLE$_{\mathcal{ZI}}(n,t)$ *terminates and preserves well-labeledness and feasibility of* $\mathcal{G}$*; moreover, it returns* false *iff* t *is clock-feasible from* n*, and ensures that* t *is disabled from* $[\![s_n]\!]$ *otherwise.*

Proof Termination of the procedure is trivial. Well-labeledness and feasibility follow from the total correctness of BLOCK. Let π be the path induced by n. Notice that $Z = post_\pi^C(Z_0)$. Assume $post_t^C(Z) \neq \bot$. Then by definition, t is clock-feasible from n, and the procedure returns false. Now assume $post_t^C(Z) = \bot$. Then by definition, t is not clock-feasible from n. By Lemma 4, we get $Z \cap pre_t^C(\top) \subseteq \bot$. Thus BLOCK$(n, pre_t^C(\top))$ can be called, and as a result, $W \cap pre_t^C(\top) \subseteq \bot$. By Lemma 4, we get $post_t^C(W) = \bot$. Thus t becomes disabled from $(\!|W|\!)$, and the procedure returns true. □

4.2.3 Interpolation for zones

The proposed refinement strategies for zone abstraction, and in particular, the different implementations of BLOCK are based on interpolation, defined over zones expressed in terms of canonical DBMs.

Definition 20 (*Zone interpolant*) Given zones A and B such that $A \cap B \subseteq \bot$, a zone interpolant is a zone I such that $A \subseteq I$ and $I \cap B \subseteq \bot$ and I is defined over the clocks that appear in both A and B.

This definition of a zone interpolant is analogous to the definition of an interpolant in the usual sense [26]. As zones correspond to formulas in $\mathcal{DL}(\mathbb{Q})$, a theory that admits interpolation [13], an interpolant always exists for a pair of disjoint zones. Algorithm 5 is a direct adaptation of the graph-based algorithm of [13] for DBMs. For simplicity, we assume that A and B are defined over the same set of clocks with the same ordering, and are both canonical (naturally, these restrictions can be lifted).

After checking the trivial cases, the algorithm searches for a negative cycle in $\min(A, B)$ to witness its inconsistency. This can be done e.g. by running a variant of the Floyd-Warshall algorithm.The interpolant I is then the DBM induced by the constraints in the negative cycle that come from A. It is easy to verify that I is indeed an interpolant.

Algorithm 5 Interpolation for Canonical DBMs

require $A \cap B \subseteq \bot$
ensure I is a zone interpolant for A and B
function INTERPOLATE$_{\mathcal{Z}}(A, B)$ **returns** I
  if $A \subseteq \bot$ **then**
    return $\bot$
  else if $B \subseteq \bot$ **then**
    return $\top$
  else
    let $M = \min(A, B)$
    let $C = \{(i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_1)\}$ be a negative cycle in M
    let $C_A = \{(i, j) \in C \mid A_{i,j} = M_{i,j}\}$
    let $I_{i,j} = \begin{cases} (0, \leq) & \text{if } i = j \\ A_{i,j} & \text{if } (i, j) \in C_A \\ \infty & \text{otherwise} \end{cases}$
    let $I = [I_{i,j}]$
    return I

Proposition 12 *Function* INTERPOLATE$_{\mathcal{Z}}$ *is totally correct: if* $A \cap B \subseteq \bot$, *then* INTERPOLATE$_{\mathcal{Z}}(A, B)$ *terminates and ensures* $A \subseteq I$ *and* $I \cap B \subseteq \bot$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Proof Function INTERPOLATE$_{\mathcal{Z}}$ has no side effect, it thus trivially maintains feasibility and well-labeledness. In the trivial cases, I is clearly an interpolant. Assume $A \neq \bot$ and $B \neq \bot$. As $A \cap B \subseteq \bot$ by contract, there exists a negative cycle C in $\min(A, B)$ by Proposition 2. As A is canonical, we can assume that no two edges are subsequent in C_A, thus the DBM I induced by C_A is clearly canonical. The properties of an interpolant directly follow from the definitions of C_A and I. □

4.2.4 Abstraction refinement for lazy zone abstraction

To maintain well-labeledness, procedures COVER and DISABLE rely on a procedure BLOCK that performs abstraction refinement by safely adjusting labels of nodes. Algorithm 6 describes two methods for abstraction refinement based on interpolation for zones. Both methods are based on pre- and post-image computation, and can be considered as a generalization of zone interpolation to sequences of transitions of a timed automaton. The main difference between the two strategies is that BLOCK$_{\text{FW}}$ (which we refer to as the "forward" zone interpolation strategy) propagates the interpolant forward using $post^C$; whereas BLOCK$_{\text{BW}}$ (which we refer to as the "backward" zone interpolation strategy) propagates "bad" zones, obtained as the complement of the interpolant, backward using pre^C.

Algorithm 6 Refinement Strategies for Lazy Zone Abstraction

```
1: ensure W ⊆ I
2: ensure I ∩ B ⊆ ⊥
3: function BLOCK_FW(n, B) returns I
4:     if W ∩ B ⊆ ⊥ then
5:         return W
6:     else
7:         if (m, n) ∈ E for some m then
8:             let t = t_(m,n)
9:             let B' = pre_t^C(B)
10:            let A' = BLOCK_FW(m, B')
11:            let A = post_t^C(A')
12:        else
13:            let A = Z
14:        let I = INTERPOLATE_Z(A, B)
15:        UPDATE(n, (Z, W ∩ I))
16:        return I

17: procedure BLOCK_BW(n, B)
18:    if W ∩ B ⊆ ⊥ then
19:        return
20:    else
21:        let I = INTERPOLATE_Z(Z, B)
22:        if (m, n) ∈ E for some m then
23:            let t = t_(m,n)
24:            for all B'' ∈ ¬I do
25:                let B''' = pre_t^C(B'')
26:                BLOCK_BW(m, B''')
27:        UPDATE(n, (Z, W ∩ I))
```

In order to make proofs of correctness for the two refinement strategies more concise, we state the following simple lemmas.

Lemma 5 $post_t^C(Z) \subseteq Z' \Rightarrow post_t(\!|Z|\!) \subseteq (\!|Z'|\!)$

Lemma 6 $(\!|Z \cap Z'|\!) = (\!|Z|\!) \cap (\!|Z'|\!)$

Proposition 13 BLOCK$_{\text{FW}}$ *is totally correct: if* $Z \cap B \subseteq \bot$, *then* BLOCK$_{\text{FW}}(n, B)$ *terminates and ensures* $W \subseteq I$ *and* $I \cap B \subseteq \bot$ *and* $W \cap B \subseteq \bot$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Proof Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract (Algorithm 4), $Z \cap B \subseteq \bot$ is ensured. Moreover, notice that $W \cap B \subseteq \bot$ follows from $W \subseteq I$ and $I \cap B \subseteq \bot$, thus it is sufficient to establish the latter two claims.

If $W \cap B \subseteq \bot$, then $I = W$, so $W \subseteq I$ and $I \cap B \subseteq \bot$ are trivially established. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise, if n is the root, then $A = Z$. Thus INTERPOLATE$_{\mathcal{Z}}(A, B)$ can be called, and the resulting interpolant I is such that $Z \subseteq I$ and $I \cap B \subseteq \bot$. As in this case $Z = Z_0$, clearly $\Sigma_0 \subseteq (\!|I|\!)$. Thus $\Sigma_0 \subseteq (\!|W \cap I|\!)$ by *initiation* and Lemma 6. Therefore, UPDATE$(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$.

Otherwise, there exists a transition $t = t_{m,n}$ for some node m. Since $Z = post_t^C(Z')$ and $B' = pre_t^C(B)$, we have $Z' \cap B' \subseteq \bot$ for $(Z', W') = s_m$ by Lemma 4. Thus BLOCK$_{\text{FW}}(m, B')$ can be called, and as a result, A' is such that $W' \subseteq A'$ and $A' \cap B' \subseteq \bot$ by contract. As $A = post_t^C(A')$, we obtain $A \cap B \subseteq \bot$ by Lemma 4. Thus INTERPOLATE$_{\mathcal{Z}}(A, B)$ can be called, and the resulting interpolant I is such that $A \subseteq I$ and $I \cap B \subseteq \bot$. By the monotonicity of images in $\subseteq$, we have $post_t^C(W') \subseteq A$. Hence $post_t^C(W') \subseteq I$, from which $post_t(\!|W'|\!) \subseteq (\!|I|\!)$ follows by Lemma 5. Thus $post_t(\!|W'|\!) \subseteq (\!|W \cap I|\!)$ by *consecution* and Lemma 6. Therefore, UPDATE$(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$. □

Proposition 14 BLOCK$_{\text{BW}}$ *is totally correct: if* $Z \cap B \subseteq \bot$, *then* BLOCK$_{\text{BW}}(n, B)$ *terminates and ensures* $W \cap B \subseteq \bot$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Proof Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract, $Z \cap B \subseteq \bot$ is ensured.

If $W \cap B \subseteq \bot$, then the contract is trivially satisfied. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise, $\text{INTERPOLATE}_{\mathcal{Z}}(Z, B)$ can be called, and the resulting interpolant I is such that $Z \subseteq I$ and $I \cap B \subseteq \bot$. We show that at the end of the procedure, the claim $W \subseteq I$, and thus $W \cap B \subseteq \bot$ holds.

Assume n is the root node. In this case $Z = Z_0$, thus clearly $\Sigma_0 \subseteq (\!|I|\!)$. Thus $\Sigma_0 \subseteq (\!|W \cap I|\!)$ by *initiation* and Lemma 6. Therefore, $\text{UPDATE}(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$.

Now assume there exists a transition $t = t_{m,n}$ for some node m with $(Z', W') = s_m$. Let $B' \in \neg I$, and $B'' = pre_t^C(B')$. Clearly, $Z \cap B' \subseteq \bot$. As $Z = post_t^C(Z')$, we obtain $Z' \cap B'' \subseteq \bot$ by Lemma 4. Thus $\text{BLOCK}_{\text{BW}}(m, B'')$ can be called, which ensures $W' \cap B'' \subseteq \bot$ by contract. Thus $post_t^C(W') \cap B' \subseteq \bot$ by Lemma 4. Hence $post_t^C(W') \subseteq I$, from which $post_t(\!|W'|\!) \subseteq (\!|I|\!)$ follows by Lemma 5. Thus $post_t(\!|W'|\!) \subseteq (\!|W \cap I|\!)$ by *consecution* and Lemma 6. Therefore, $\text{UPDATE}(n, (Z, W \cap I))$ can be called, which establishes $W \subseteq I$, while preserving the well-labeledness of $\mathcal{G}$. □

We would like to point out that for refinement with BLOCK_{FW}, syntactically, it is sufficient to store a single zone at each node, thus obtaining a major optimization in memory consumption. In particular, it is sufficient to store Z at leaves, and store W at non-leaf nodes. This is due to the fact that while running the algorithm, Z is only necessary when EXPAND is called, and when the interpolant is computed for the initial node, in this later situation Z being obvious. On the other hand, W is only necessary when calling COVER, where coverer nodes are always non-leaf. Moreover, it is always safe to treat W as $\bot$ for leafs.

4.3 Abstraction and refinement for discrete variables

In the following, we describe strategies for the handling of discrete variables that appear in timed automata specifications.

4.3.1 Explicit tracking of variables

The most straightforward way for the handling discrete variables is to explicitly track their value.

Definition 21 (*Explicit domain*) Let $\mathcal{E} = \mathcal{V}(D)$. We define the abstraction that tracks discrete variables explicitly as the abstract domain $\mathbb{D}_{\mathcal{E}} = (\mathcal{E}, =, \nu_0, post^D, (\!|\cdot|\!))$.

Proposition 15 *$\mathbb{D}_{\mathcal{E}}$ is sound.*

Similarly to zone abstraction, we define $\text{COVER}_{\mathcal{E}}$ to be a no-op, thus its total correctness is trivial. Moreover, let $\text{DISABLE}_{\mathcal{E}}(n, t) \stackrel{\circ}{=} (\mathsf{post}_t(\nu) \sqsubseteq \bot)$ where $\nu = s_n$.

Proposition 16 $\text{DISABLE}_{\mathcal{E}}$ *is totally correct:* $\text{DISABLE}_{\mathcal{E}}(n, t)$ *terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns* **false** *iff t is data-feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof Termination of the procedure is trivial. Well-labeledness and feasibility follow from the fact that the procedure has no side effects. Let π be the path induced by n. Notice that $\nu = post^D_\pi(\nu_0)$. Assume $post^D_t(\nu) \neq \bot$. Then by definition, t is data-feasible from n, and the procedure returns false. Now assume $post^D_t(\nu) = \bot$. Then by definition, t is not data-feasible from n. But t is also disabled from $(\!|\nu|\!)$, and the procedure returns true. □

4.3.2 Visible variables abstraction

Instead of explicitly tracking in all states the values for all variables, by tracking in each state only those that play a role in unreachability of a given location along a path through the state, and "hiding" all the others, the size of the explored state space can be significantly reduced. In the following, we describe such an abstract domain, together with the corresponding refinement strategies.

Definition 22 (*Visible variables domain*) Let $\mathbb{D}_{\mathcal{EI}} = (\mathcal{S}, \sqsubseteq, \mathsf{init}, \mathsf{post}, [\![\cdot]\!])$ be the abstract domain over $\mathbb{D}_{\mathcal{E}}$ with

- $\mathcal{S} = \mathcal{V}(D) \times \mathcal{P}(D)$,
- $(\nu, Q) \sqsubseteq (\nu', Q')$ iff $\nu \preceq \nu'{\restriction_{Q'}}$ and $Q' \subseteq Q$ (thus $\sqsubseteq$ is a preorder),
- $\mathsf{init} = (\mathsf{init}, \emptyset)$,
- $\mathsf{post}_t(\nu, Q) = (\mathsf{post}_t(\nu), \emptyset)$, and
- $[\![(\nu, Q)]\!] = (\!|\nu{\restriction_Q}|\!)$.

Proposition 17 *$\mathbb{D}_{\mathcal{EI}}$ is sound.*

Algorithm 7 describes the corresponding refinement methods. Both $\textsc{cover}_{\mathcal{EI}}$ and $\textsc{disable}_{\mathcal{EI}}$ rely on a procedure REFINE for abstraction refinement. Moreover, $\textsc{disable}_{\mathcal{EI}}$ depends on a weakest precondition operator, defined by the following property.

Definition 23 (*Weakest discrete precondition*) Let $wp^D_t(\varphi)$ be the formula such that $\nu \models wp^D_t(\varphi)$ iff $post^D_t(\nu) \models \varphi$ for all ν and φ, with respect to t.

In $\textsc{cover}_{\mathcal{EI}}$, as $\nu \preceq \nu'{\restriction_{Q'}}$, we have $\nu \models \mathsf{form}(\nu'{\restriction_{Q'}})$ by Lemma 2, thus calling $\textsc{refine}(n, \mathsf{form}(\nu'{\restriction_{Q'}}))$ is safe. Other than that, total correctness of $\textsc{cover}_{\mathcal{EI}}$ follows trivially from total correctness of REFINE (see later).

Proposition 18 $\textsc{disable}_{\mathcal{EI}}$ *is totally correct:* $\textsc{disable}_{\mathcal{EI}}(n, t)$ *terminates and preserves well-labeledness and feasibility of $\mathcal{G}$; moreover, it returns* false *iff t is data-feasible from n, and ensures that t is disabled from $[\![s_n]\!]$ otherwise.*

Proof Termination of the procedure is trivial. Well-labeledness and feasibility follow from the total correctness of REFINE. Let π be the path induced by n. Notice that $\nu = post^D_\pi(\nu_0)$. Assume $post^D_t(\nu) \neq \bot$. Then by definition, t is data-feasible from n, and the procedure returns false. Now assume $post^D_t(\nu) = \bot$. Then by definition, t is not data-feasible from n. As $post^D_t(\nu) \models \bot$, by Definition 23, we get $\nu \models wp^D_t(\bot)$. Thus $\textsc{refine}(n, wp^D_t(\bot))$ can be called, and as a result, $\nu{\restriction_Q} \models wp^D_t(\bot)$. By Definition 23, we get $post^D_t(\nu{\restriction_Q}) \models \bot$, thus clearly $post^D_t(\nu{\restriction_Q}) = \bot$. Thus t becomes disabled from $(\!|\nu{\restriction_Q}|\!)$, and the procedure returns true. □

Algorithm 7 Visible Variables Abstraction

```
1: procedure COVER_{E_I}(n, n')
2:     let (ν, ·) = s_n
3:     let (ν', Q') = s_{n'}
4:     if ν ⪯ ν'↑_{Q'} then
5:         REFINE(n, form(ν'↑_{Q'}))
6: function DISABLE_{E_I}(n, t)
7:     let (ν, ·) = s_n
8:     let ν' = post_t^D(ν)
9:     if ν' = ⊥ then
10:        REFINE(n, wp_t^D(⊥))
11:        return true
12:    else
13:        return false

14: invariant G is well-labeled and feasible
15: define (ν, Q) = s_n
16: require ν ⊨ φ
17: ensure ν↑_Q ⊨ φ
18: procedure REFINE(n, φ)
```

4.3.3 Interpolation for valuations

The proposed refinement strategies for discrete variables, and in particular, different implementations of REFINE are based on the notion of a valuation interpolant, defined over a valuation and a formula.

Definition 24 (*Valuation interpolant*) Given a valuation σ and a formula φ such that $\sigma \models \varphi$, a valuation interpolant is a valuation σ' such that $\sigma \preceq \sigma'$ and $\sigma' \models \varphi$ and $\mathsf{def}(\sigma') \subseteq \mathsf{def}(\sigma) \cap \mathsf{vars}(\varphi)$.

Algorithm 8 Interpolation for Valuations

```
1: invariant G is well-labeled and feasible
2: require σ ⊨ φ
3: ensure σ↑_I is an interpolant for σ and φ
4: function INTERPOLATE_E(σ, φ) returns I
5:     let X = def(σ) ∩ vars(φ)
6:     I ← X
7:     for all x ∈ X do
8:         let I' = I \ {x}
9:         if σ↑_{I'} ⊨ φ then
10:            I ← I'
11:    return I
```

Proposition 19 *Function* INTERPOLATE$_{\mathcal{E}}$ *is totally correct: if* $\sigma \models \varphi$, *then* INTERPOLATE$_{\mathcal{E}}(\sigma, \varphi)$ *terminates and ensures* $\sigma{\uparrow_I} \models \varphi$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Proof Function INTERPOLATE$_{\mathcal{E}}$ has no side effect, it thus trivially maintains feasibility and well-labeledness. Moreover, it is easy to see that it satisfies its contract, as the postcondition is an invariant for the loop. □

Next, we show how valuation interpolants can be used for hiding variables that are irrelevant with respect to the reachability of a given location along a path.

4.3.4 Abstraction refinement for visible variables abstraction

Algorithm 9 outlines two strategies for abstraction refinement over the visible variables abstract domain. Symmetrically to the variants of BLOCK, procedure $\text{REFINE}_{\text{FW}}$ (which we refer to as the "forward" valuation interpolation strategy) propagates interpolants forward using $post^D$; whereas procedure $\text{REFINE}_{\text{BW}}$ (which we refer to as the "backward" valuation interpolation strategy) propagates interpolants backward using wp^D along the path to be refined.

Algorithm 9 Refinement Strategies for Visible Variables Abstraction

```
1: ensure I ⊆ Q
2: ensure ν↾_I ⊨ φ
3: function REFINE_FW(n, φ) returns ν↾_I
4:     if ν↾_Q ⊨ φ then
5:         return ν↾_Q
6:     else
7:         if (m, n) ∈ E for some m then
8:             let t = t_(m,n)
9:             let φ' = wp_t^D(φ)
10:            let α' = REFINE_FW(m, φ')
11:            let α = post_t^D(α')
12:        else
13:            let α = ν
14:        let I = INTERPOLATE_E(α, φ)
15:        UPDATE(n, (ν, Q ∪ I))
16:        return ν↾_I

17: procedure REFINE_BW(n, φ)
18:     if ν↾_Q ⊨ φ then
19:         return
20:     else
21:         let I = INTERPOLATE_E(ν, φ)
22:         if (m, n) ∈ E for some m then
23:             let t = t_(m,n)
24:             let φ' = wp_t^D(form(ν↾_I))
25:             REFINE_BW(m, φ')
26:         UPDATE(n, (ν, Q ∪ I))
```

To make our formal description more concise, we state the following simple lemmas.

Lemma 7 $\alpha \preceq \beta \Rightarrow post_t^D(\alpha) \preceq post_t^D(\beta)$

Lemma 8 $post_t^D(\nu) \preceq \nu' \Rightarrow post_t(\!|\nu|\!) \subseteq (\!|\nu'|\!)$

Lemma 9 $(\!|\nu\restriction_{A\cup B}|\!) = (\!|\nu\restriction_A|\!) \cap (\!|\nu\restriction_B|\!)$

Proposition 20 $\text{REFINE}_{\text{FW}}$ *is totally correct: if* $\nu \models \varphi$, *then* $\text{REFINE}_{\text{FW}}(n, \varphi)$ *terminates and ensures* $I \subseteq Q$ *and* $\nu\restriction_I \models \varphi$ *and* $\nu\restriction_Q \models \varphi$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Proof Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract, $\nu \models \varphi$ is ensured. Moreover, notice that $\nu\restriction_Q \models \varphi$ follows from $I \subseteq Q$ and $\nu\restriction_I \models \varphi$ by Lemma 1, thus it is sufficient to establish the latter two claims.

If $\nu\restriction_Q \models \varphi$, then $I = Q$, so $I \subseteq Q$ and $\nu\restriction_I \models \varphi$ are trivially established. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise, if n is the root, then $\alpha = \nu$. Thus $\text{INTERPOLATE}_{\mathcal{E}}(\alpha, \varphi)$ can be called, and the resulting interpolant I is such that $\nu\restriction_I \models \varphi$. As in this case $\nu = \nu_0$, clearly $\Sigma_0 \subseteq (\!|\nu\restriction_I|\!)$. Thus $\Sigma_0 \subseteq (\!|\nu\restriction_{Q\cup I}|\!)$ by *initiation* and Lemma 9. Therefore, $\text{UPDATE}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$.

Otherwise, there exists a transition $t = t_{m,n}$ for some node m. Since $\nu = post_t^D(\nu')$ and $\varphi' = wp_t^D(\varphi)$, we have $\nu' \models \varphi'$ for $(\nu', Q') = s_m$ by Definition 23. Thus $\text{REFINE}_{\text{FW}}(m, \varphi')$ can

Table 1 Summary of refinement strategies

	Lazy zone interpolation		Lazy valuation interpolation	
	Forward	Backward	Forward	Backward
$\mathbb{D}$	$\mathbb{D}_{\mathcal{ZI}}$		$\mathbb{D}_{\mathcal{EI}}$	
COVER	$\text{COVER}_{\mathcal{ZI}}$		$\text{COVER}_{\mathcal{EI}}$	
DISABLE	$\text{DISABLE}_{\mathcal{ZI}}$		$\text{DISABLE}_{\mathcal{EI}}$	
Propagation	BLOCK_{FW}	BLOCK_{BW}	$\text{REFINE}_{\text{FW}}$	$\text{REFINE}_{\text{BW}}$
Interpolation	$\text{INTERPOLATE}_{\mathcal{Z}}$		$\text{INTERPOLATE}_{\mathcal{E}}$	

be called, and as a result, α' is such that $\alpha' = \nu'\restriction_{I'}$ and $I' \subseteq Q'$ and $\alpha' \models \varphi'$ by contract for some I'. As $\alpha = post_t^D(\alpha')$, we obtain $\alpha \models \varphi$ by Definition 23. Thus $\text{INTERPOLATE}_{\mathcal{E}}(\alpha, \varphi)$ can be called, and the resulting interpolant I is such that $\alpha\restriction_I \models \varphi$. Clearly $\nu' \preceq \alpha'$, thus $\nu \preceq \alpha$ by Lemma 7. Therefore, $\nu\restriction_I = \alpha\restriction_I$, as $I \subseteq \mathsf{def}(\alpha)$. From this, $\nu\restriction_I \models \varphi$ follows directly. Moreover, as $\nu'\restriction_{Q'} \preceq \nu'\restriction_{I'}$, by Lemma 7, we have $post_t^D(\nu'\restriction_{Q'}) \preceq \alpha$. Hence $post_t^D(\nu'\restriction_{Q'}) \preceq \nu\restriction_I$, from which $post_t(\!|\nu'\restriction_{Q'}|\!) \subseteq (\!|\nu\restriction_I|\!)$ follows by Lemma 8. Thus $post_t(\!|\nu'\restriction_{Q'}|\!) \subseteq (\!|\nu\restriction_{Q\cup I}|\!)$ by *consecution* and Lemma 9. Therefore, $\text{UPDATE}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$. □

Proposition 21 $\text{REFINE}_{\text{BW}}$ *is totally correct: if* $\nu \models \varphi$, *then* $\text{REFINE}_{\text{BW}}(n, \varphi)$ *terminates and ensures* $\nu\restriction_Q \models \varphi$. *Moreover, it preserves well-labeledness and feasibility of* $\mathcal{G}$.

Proof Termination of the procedure is trivial. Moreover, the procedure trivially maintains feasibility of $\mathcal{G}$, as it does not create new nodes. Thus we focus on partial correctness and the preservation of well-labeledness. By contract, $\nu \models \varphi$ is ensured.

If $\nu\restriction_Q \models \varphi$, then the contract is trivially satisfied. Moreover, well-labeledness is trivially maintained, as no refinement is performed.

Otherwise $\text{INTERPOLATE}_{\mathcal{E}}(\nu, \varphi)$ can be called, and the resulting interpolant I is such that $\nu\restriction_I \models \varphi$. We show that at the end of the procedure, the claim $I \subseteq Q$, and thus by Lemma 1 also $\nu\restriction_Q \models \varphi$ holds.

Assume n is the root node. In this case $\nu = \nu_0$, thus clearly $\Sigma_0 \subseteq (\!|\nu\restriction_I|\!)$. Thus $\Sigma_0 \subseteq (\!|\nu\restriction_{Q\cup I}|\!)$ follows by *initiation* and Lemma 9. As a consequence, $\text{UPDATE}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$.

Now assume there exists a transition $t = t_{m,n}$ for some node m with $(\nu', Q') = s_m$. Clearly, $\nu \preceq \nu\restriction_I$, thus $\nu \models \mathsf{form}(\nu\restriction_I)$ by Lemma 2. As $\nu = post_t^D(\nu')$ and $\varphi' = wp_t^D(\mathsf{form}(\nu\restriction_I))$ we obtain $\nu' \models \varphi'$ by Definition 23. Thus $\text{REFINE}_{\text{BW}}(m, \varphi')$ can be called, which ensures $\nu'\restriction_{Q'} \models \varphi'$ by contract. Thus $post_t^D(\nu'\restriction_{Q'}) \models \mathsf{form}(\nu\restriction_I)$ by Definition 23. Hence $post_t^D(\nu'\restriction_{Q'}) \preceq \nu\restriction_I$ by Lemma 2, from which $post_t(\!|\nu'\restriction_{Q'}|\!) \subseteq (\!|\nu\restriction_I|\!)$ follows by Lemma 8. Thus $post_t(\!|\nu'\restriction_{Q'}|\!) \subseteq (\!|\nu\restriction_{Q\cup I}|\!)$ by *consecution* and Lemma 9. As a consequence, $\text{UPDATE}(n, (\nu, Q \cup I))$ can be called, which establishes $I \subseteq Q$, while preserving the well-labeledness of $\mathcal{G}$. □

Finally, to conclude this section, Table 1 summarizes the different abstraction refinement strategies discussed.

5 Evaluation

In this section, we investigate how each algorithm configuration that our framework supports behaves on a wide range of timed automata models in terms of performance (execution time)

and size of the state space (number of nodes in the generated ART). The input models, raw measurement data, and instructions to reproduce our experiments are available in a supplementary material [32].

We implemented a prototype version of our algorithm and refinement strategies in the open source model checking framework THETA [29]. Our tool performs location reachability checking on models given in a reasonable language subset[1] of the UPPAAL 4.0 XTA format.

To enable comparison to the state-of-the-art, we implemented in our framework a variant of the lazy abstraction method of [22] based on *LU*-bounds as an alternative refinement strategy for clock variables (by defining the domain, COVER and DISABLE accordingly). The main difference in our implementation compared to [22] is that when performing abstraction refinement, bounds are propagated from all guards on an infeasible path, and not just from ones that contribute to the infeasibility. Because of this, refinement in the resulting algorithm is extremely cheap, but as the comparison of our data with that of [22] suggests, for the models examined in both papers, the algorithm is similarly as space- and time-efficient as the original one.

The algorithms are evaluated for both breadth-first and depth-first search orders of ART expansion. By combining all the possible alternatives, this results in 16 distinct algorithm configurations:

- as search order, breadth-first (BFS) or depth-first (DFS) search,
- for clock variables, forward (FWITP) or backward (BWITP) zone interpolation, or lazy $\mathfrak{a}_{\preceq_{LU}}$ abstraction (LU),
- for discrete variables, forward (FWITP) or backward (BWITP) valuation interpolation, or no refinement (NONE).

Each algorithm configuration is encoded as a string containing three characters, specifically the first character of the name of each selected parameter. So for example, the configuration with BFS as search order, LU as refinement strategy for clock variables, and NONE as refinement strategy for discrete variables, is going to be encoded as BLN.

For the configurations that handle discrete variables explicitly (·N·), we partitioned the set of nodes of the ART 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 51 timed automata models in total, which we divided to three distinct categories. For each model, the number of clock variables/number of discrete variables is given in parentheses.

- Category PAT: classic timed automata models from the PAT benchmark set.[2] These models contain only a few discrete variables.
  - critical n with $n \in \{3, 4\}$ (n/1): Critical Region with n processes.
  - csma n with $n \in \{9, 10, 11, 12\}$ (n/1): CSMA/CD protocol with n processes.
  - fddi n with $n \in \{50, 70, 90, 110\}$ ($3n + 1$/1): FDDI token ring with n processes.
  - fischer n with $n \in \{7, 8, 9, 10\}$ (n/1): Fischer's mutual exclusion protocol with n processes.
  - lynch n with $n \in \{7, 8, 9\}$ (n/2): Lynch-Shavit protocol with n processes.

[1] Not supporting procedures and composite types other than arrays of synchronization channels.

[2] https://www.comp.nus.edu.sg/~pat/bddlib/timedexp.html.

– Category MCTA: model containing a significant number of discrete variables (relative to the number of clock variables). Most of the models come from the MCTA benchmark set,[3] while some of them come from the UPPAAL benchmark set.[4]
  – bocdp (3/26), bocdpf (3/26): models of the Bang & Olufsen Collision Detection Protocol obtained from the UPPAAL benchmark set.
  – brp (7/7): a model of the Bounded Retransmission Protocol.
  – c1 (3/12), c2 (3/14), c3 (3/15), c4 (3/17): models of a real-time mutual exclusion protocol obtained from the MCTA benchmark set.
  – m1 (4/11), m2 (4/13), m3 (4/13), m4 (4/15), n1 (7/11), n2 (7/13), n3 (7/13), n4 (7/15), e1 (3/41): industrial cases studies obtained from the MCTA benchmark set.

– Fischer's protocol with diagonal constraints, based on [28]
  – diag n with $n \in \{3, 4, 5, 6, 7, 8\}$ $(2n/1)$: the original model, containing diagonal constraints.
  – split n with $n \in \{3, 4, 5, 6, 7, 8\}$ $(2n/n+1)$: diagonal-free model obtained from diag n by eliminating diagonal constraints by introducing additional discrete variables and transitions, following the idea described in [6].
  – opt n with $n \in \{3, 4, 5, 6, 7, 8\}$ $(2n/n+1)$: diagonal-free model obtained from split n by (manually) removing some guards, updates and transitions about which it can statically be established that they do not influence the set of reachable locations.

We performed our measurements on a machine running Windows 10 with a 2.6 GHz dual core CPU and 8GB of RAM. We evaluated the algorithm configurations for both execution time and the number of nodes in the resulting ART. The timeout (denoted by "–" in the tables) was set to 300 s. The execution time shown in the following tables is the average of 10 runs, obtained from 12 deterministic runs by removing the slowest and the fastest one. For each model, the value belonging to the best configuration is typeset in **bold**.

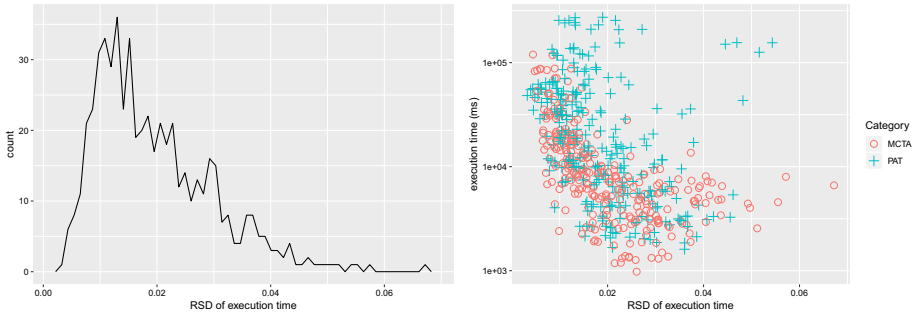
5.1 Diagonal-free models

Performing location reachability checking on the models, Figure 1a shows the frequency with which different relative standard deviation (RSD) values of execution time occur. It can be seen from the plot that higher RSD values ($> 5\%$) are relatively rare among the measurements. Moreover, Figure 1b shows how the RSD of execution time relates to the average execution time for each model and configuration (in this type of figures, each point represents the average result for a given model and configuration). Aside from a few outliers among the PAT models, it can be stated that higher RSD values belong to small average execution times, as expected. Thus it is justifiable to base the comparison of configurations on the average value.

As can be seen on Figure 2, on the selected benchmark set, having all other configuration parameters fixed, clock refinement strategies FWITP and BWITP do not significantly differ in performance. On both benchmarks, FWITP slightly outperforms BWITP in the size of the generated state space. Moreover, for the MCTA models, FWITP, while for the PAT models, BWITP performs slightly better in terms of execution time (note the logarithmic scale on

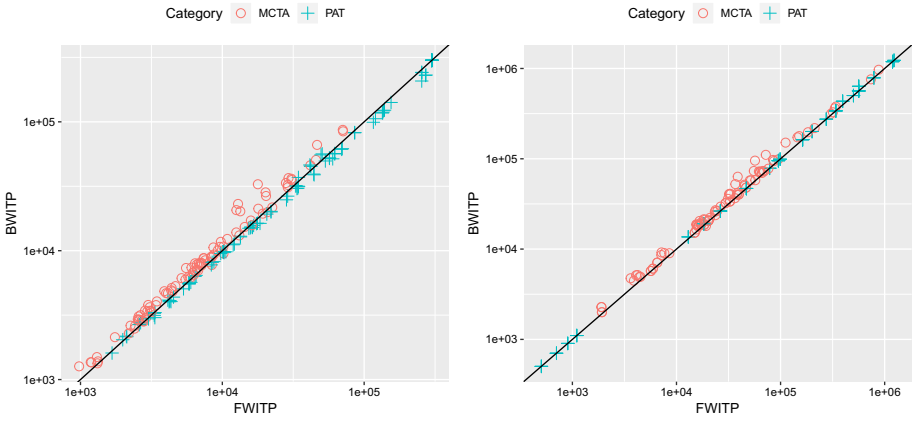
[3] http://gki.informatik.uni-freiburg.de/tools/mcta/benchmarks.html.

[4] https://www.it.uu.se/research/group/darts/uppaal/benchmarks.



(a) Frequency **(b)** Compared to Execution Time (ms)

Fig. 1 Relative Standard Deviation of Execution Time



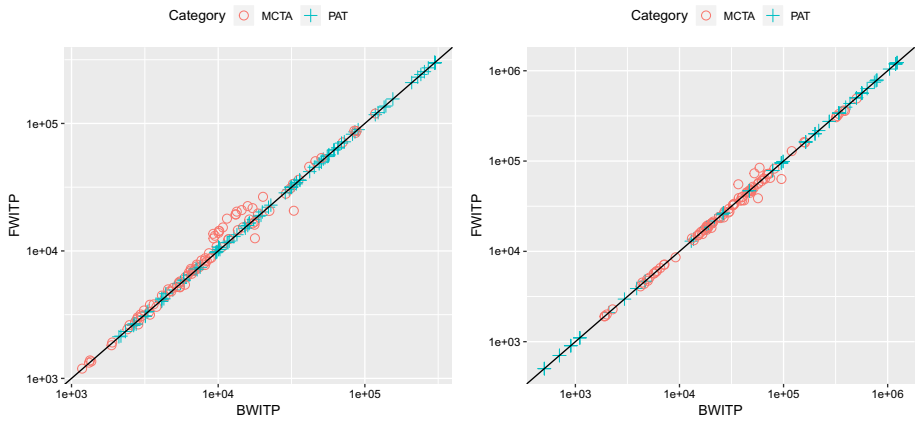
(a) Execution Time (ms) **(b)** Number of Nodes

Fig. 2 Clock Refinement: BWITP vs. FWITP

the axes). An explanation for this is that in general, FWITP tends to perform less refinement steps (as refinement is performed in a single iteration), whereas BWITP performs refinement steps more cheaply (as no post-image computation is involved). In our experiments, the two algorithms performed roughly the same number of refinement steps for the PAT models (probably due to discovering the same or similar simple invariants), in which case BWITP has an advantage. In the case of MCTA models however, in general, the number of refinement steps performed was in favor of FWITP.

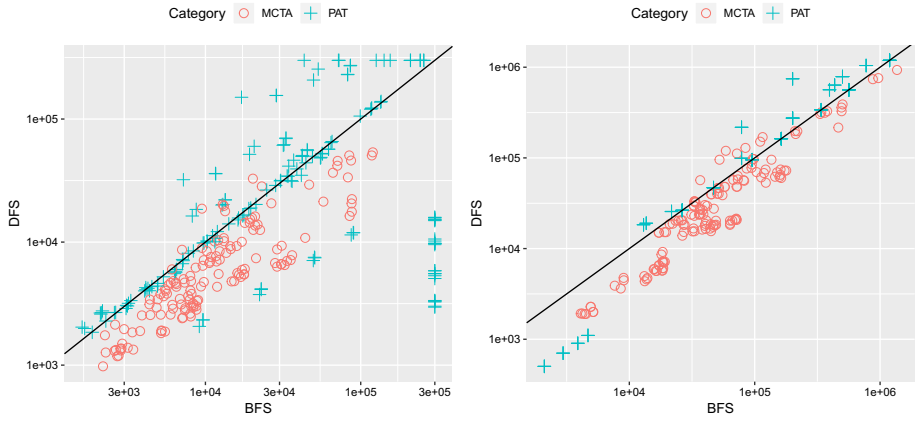
Figure 3 shows similar results for comparing the two interpolation-based strategies for discrete refinement. Here, BWITP tends to perform better in terms of execution time. Therefore, we are going to omit detailed results for clock refinement BWITP and discrete refinement FWITP for the rest of the section.

Figure 4 compares the impact of the two search orders on performance. With respect to execution time, DFS generally outperforms BFS on the MCTA models, whereas on the PAT models, the performance of the two search orders is balanced. When considering the size of the state space, the tendency is similar.



(a) Execution Time (ms) **(b)** Number of Nodes

Fig. 3 Discrete Refinement: FWITP vs. BWITP



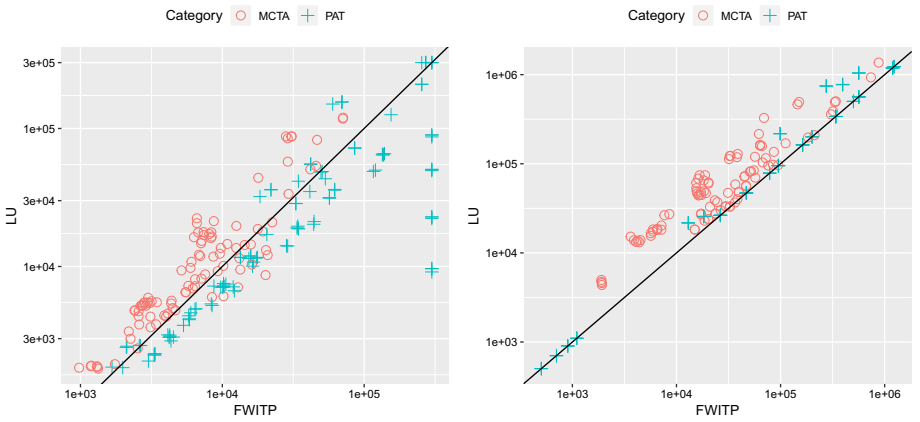
(a) Execution Time (ms) **(b)** Number of Nodes

Fig. 4 Search Order: DFS vs. BFS

Clock refinements LU and FWITP are compared on Figure 5. With respect to execution time, LU performs better in category PAT, whereas FWITP performs better in category MCTA. However, with respect to the size of the state space, FWITP outperforms LU.

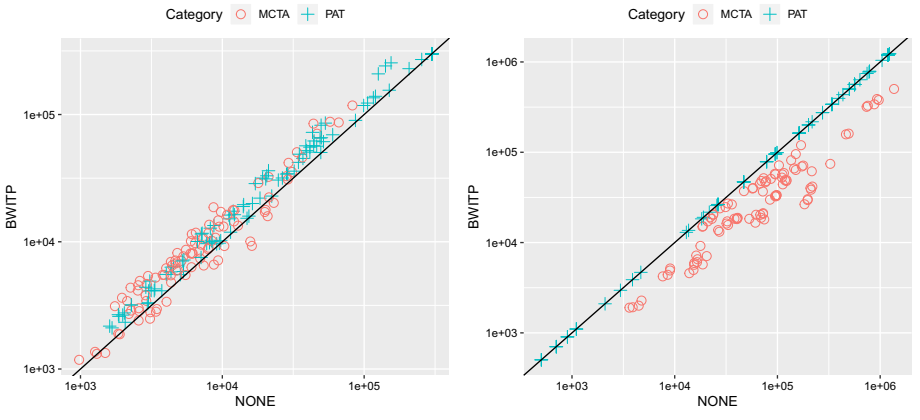
Figure 6 shows the pairwise comparison of interpolation-based and explicit handling of discrete variables. On the MCTA models, BWITP is always able to generate an—in some cases, significantly—smaller state space. Unsurprisingly, the same reduction effect is not present on PAT models, where there are only one or two discrete variables. Despite the significant reduction in state space, on the models considered, aside from a couple of cases, BWITP is somewhat slower. Beside the obvious overhead of running abstraction refinement, this can be explained with the optimization of coverage checking applied in the explicit case, as described above.

The detailed results for the PAT models are shown in Table 2. On these models, configurations BLN and DLN usually perform best in terms of execution time. When considering the size of the state space however, there is a small variability between configurations. Moreover,



(a) Execution Time (ms) **(b)** Number of Nodes

Fig. 5 Clock Refinement: LU vs. FWITP



(a) Execution Time (ms) **(b)** Number of Nodes

Fig. 6 Discrete Refinement: BWITP vs. NONE

we point our that our results for configurations BLN and DLN are consistent with the results presented in [22].

Detailed results for the MCTA models are shown in Table 3. Here, configurations DFN or DFB give the fastest execution on most models. Moreover, configuration DFB generates the least number of nodes in almost all cases, which highlights the advantages of our new interpolation based algorithm presented first in [31].

5.2 Models containing diagonal constraints

We also evaluated how the different configurations are able to handle models with diagonal constraints. As our benchmark, we used the diagonal version of Fischer's mutual exclusion algorithm, as presented in [28]. We considered two approaches:

Table 2 Detailed results for PAT models

Model	BFB	BFN	BLB	BLN	DFB	DFN	DLB	DLN
(a) Execution time (s)								
critical 3	2.1	**1.7**	2.6	1.9	2.7	2.0	2.7	1.8
critical 4	42.1	**34.4**	55.4	41.4	50.6	41.4	48.8	34.9
csma 9	13.4	8.7	11.7	**7.2**	22.0	18.4	35.9	32.1
csma 10	33.0	20.6	28.7	**17.1**	69.3	60.0	155.2	150.3
csma 11	85.6	53.2	72.4	**43.2**	270.6	254.7	–	–
csma 12	254.7	154.8	208.7	**125.8**	–	–	–	–
fddi 50	–	–	9.6	9.1	3.3	3.0	2.3	**2.1**
fddi 70	–	–	22.9	22.3	5.8	5.3	4.1	**3.7**
fddi 90	–	–	50.3	49.5	10.2	9.7	7.5	**7.1**
fddi 110	–	–	90.0	86.8	15.9	15.4	11.9	**11.4**
fischer 7	4.1	3.3	3.2	2.3	4.3	3.3	3.2	**2.3**
fischer 8	10.3	8.4	7.2	5.4	10.1	8.5	7.1	**5.2**
fischer 9	34.1	28.3	19.6	14.1	34.2	28.9	18.9	**14.1**
fischer 10	135.3	116.1	65.4	**48.9**	139.1	120.1	65.7	49.9
lynch 7	6.4	4.5	4.9	3.1	5.8	4.3	4.4	**2.9**
lynch 8	17.4	11.9	11.5	7.1	16.3	12.2	10.1	**6.7**
lynch 9	62.0	44.4	36.2	21.2	56.9	44.1	31.2	**20.2**

Table 2 continued

Model	BFB	BFN	BLB	BLN	DFB	DFN	DLB	DLN
(b) Number of Nodes								
critical 3	12,981	12,981	21,699	21,699	18,310	18,310	25,697	25,697
critical 4	395,188	394,525	772,221	777,784	564,014	564,014	1,043,487	1,043,487
csma 9	78,552	78,552	78,552	78,552	98,989	98,989	217,656	217,656
csma 10	200,649	200,649	200,649	200,649	274,759	274,759	745,149	745,149
csma 11	501,432	501,432	501,432	501,432	787,898	787,898	–	–
csma 12	1,230,757	1,230,757	1,230,757	1,230,757	–	–	–	–
fddi 50	–	–	2098	2098	503	503	503	503
fddi 70	–	–	2961	2961	703	703	703	703
fddi 90	–	–	3881	3881	903	903	903	903
fddi 110	–	–	4678	4678	1103	1103	1103	1103
fischer 7	26,405	26,405	26,405	26,405	26,405	26,405	26,405	26,405
fischer 8	95,353	95,353	95,353	95,353	95,353	95,353	95,353	95,353
fischer 9	339,211	339,211	339,211	339,211	339,211	339,211	339,211	339,211
fischer 10	1,191,211	1,191,211	1,191,211	1,191,211	1,191,211	1,191,211	1,191,211	1,191,211
lynch 7	46,915	46,915	46,915	46,915	46,915	46,915	46,915	46,915
lynch 8	162,801	162,801	162,801	162,801	162,801	162,801	162,801	162,801
lynch 9	563,491	563,491	563,491	563,491	563,491	563,491	563,491	563,491

Table 3 Detailed results for MCTA models

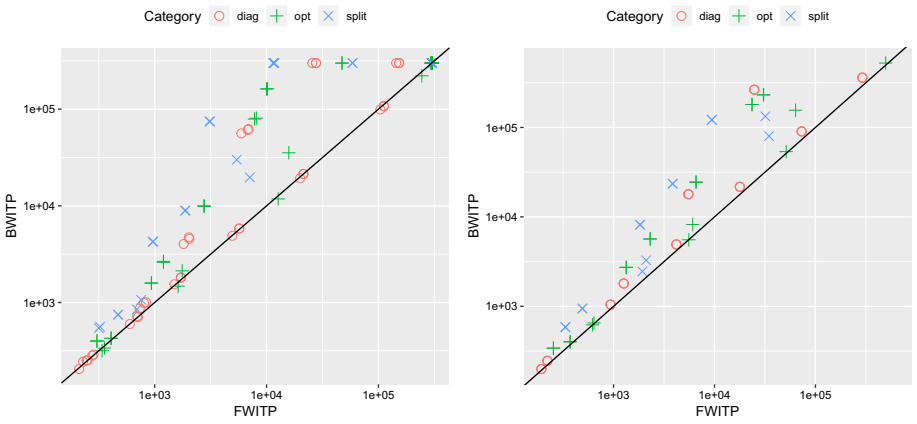
Model	BFB	BFN	BLB	BLN	DFB	DFN	DLB	DLN
(a) Execution time (s)								
bocdp	13.2	10.2	11.5	6.1	10.2	8.5	10.1	**6.0**
bocdpf	15.9	20.9	14.5	12.1	9.3	16.2	**9.3**	10.3
brp	13.4	12.9	9.5	**7.1**	17.8	20.2	18.7	8.7
c1	4.4	2.3	5.4	3.0	3.1	**1.7**	3.6	2.0
c2	8.7	5.5	11.8	6.5	6.2	**4.0**	7.0	4.3
c3	9.8	6.4	13.6	8.1	7.1	**4.7**	8.2	4.8
c4	70.7	46.6	117.8	82.7	41.7	**29.3**	50.6	33.6
e1	5.5	3.9	6.5	4.4	4.1	**2.5**	4.6	2.6
m1	2.7	2.2	5.2	3.4	1.2	**1.0**	1.9	1.8
m2	7.1	5.2	14.7	9.4	**2.4**	2.6	4.8	4.4
m3	8.1	6.0	17.2	9.8	3.0	**2.6**	5.9	4.7
m4	28.9	17.9	84.8	43.9	6.3	**6.1**	16.3	10.8
n1	2.9	2.6	5.5	3.8	1.3	**1.3**	1.9	1.9
n2	7.4	7.0	17.7	11.9	**2.8**	3.1	5.4	4.3
n3	8.4	6.8	17.7	12.2	**3.0**	3.5	5.5	5.5
n4	30.9	28.9	87.7	57.5	**6.6**	8.7	22.3	21.3
(b) Number of nodes								
bocdp	32,639	94,801	33,030	96,460	**29,846**	84,643	33,341	97,462
bocdpf	38,492	212,225	40,083	209,430	**26,544**	183,402	30,230	197,234
brp	**36,761**	72,117	58,825	115,675	56,786	111,705	119,826	169,672
c1	17,156	20,967	27,058	32,963	**14,973**	18,614	18,292	22,968
c2	44,906	67,433	71,657	103,476	**39,644**	57,170	48,069	69,760
c3	50,713	86,285	81,524	136,015	**46,593**	76,335	56,833	95,548
c4	339,560	876,266	502,423	1,365,289	**318,480**	737,964	389,018	932,334
e1	24,677	31,247	37,105	47,199	**20,299**	23,657	23,931	27,513
m1	4394	8541	13,171	27,216	**1901**	3625	4970	15,233
m2	16,246	31,932	44,095	112,634	**5673**	15,471	16,603	60,995
m3	18,369	38,128	49,032	118,485	**7181**	16,189	20,291	68,091
m4	66,255	145,378	157,864	464,477	**20,335**	61,915	61,606	215,984
n1	4222	7645	13,731	26,467	**1921**	3898	4579	13,869
n2	15,648	33,054	49,197	122,680	**5933**	15,514	18,315	53,212
n3	17,177	32,493	48,007	122,178	**6536**	16,677	18,031	74,393
n4	63,674	150,864	160,825	493,530	**18,798**	69,308	74,430	326,938

1. Eager elimination of difference constraints by introducing new discrete variables (models split n and manually optimized versions opt n).
2. Applying abstraction refinement to the model with diagonal constraints directly (models diag n).

Table 4 shows our detailed measurement data for all three types of models. Models split n, where diagonal constraints are eliminated, enable the comparison of our approach with state-of-the-art approaches presented in [18,28]. We point out that our results for configuration

Table 4 Detailed results for the diagonal version of Fischer's protocol

Model	BFB	BFN	BLB	BLN	DFB	DFN	DLB	DLN
(a) Execution time (s)								
diag 3	0.3	**0.2**	–	–	0.3	0.2	–	–
diag 4	0.7	**0.6**	–	–	0.8	0.7	–	–
diag 5	1.7	**1.5**	–	–	2.0	1.8	–	–
diag 6	5.7	**4.9**	–	–	6.9	6.0	–	–
diag 7	21.4	**19.9**	–	–	27.7	25.7	–	–
diag 8	111.8	**104.1**	–	–	153.6	144.2	–	–
split 3	**0.3**	0.7	0.4	0.6	0.5	0.8	0.4	0.6
split 4	**1.0**	7.1	1.9	5.5	1.9	5.4	2.5	5.3
split 5	**3.1**	–	19.9	259.4	11.8	–	45.4	–
split 6	**11.6**	–	–	–	–	–	–	–
split 7	**58.5**	–	–	–	–	–	–	–
split 8	–	–	–	–	–	–	–	–
opt 3	0.3	0.3	0.3	0.2	0.4	0.4	0.3	**0.2**
opt 4	0.9	1.6	0.9	0.9	1.2	1.8	1.0	**0.8**
opt 5	**2.8**	12.7	4.3	4.8	7.8	15.8	4.5	4.1
opt 6	**10.0**	244.4	36.4	49.9	–	–	43.9	39.3
opt 7	**47.1**	–	–	–	–	–	–	–
opt 8	**293.5**	–	–	–	–	–	–	–
(b) Number of nodes								
diag 3	193	193	–	–	220	220	–	–
diag 4	933	933	–	–	1262	1262	–	–
diag 5	4181	4181	–	–	5515	5515	–	–
diag 6	17,815	17,815	–	–	24,772	24,772	–	–
diag 7	73,137	73,137	–	–	100,147	100,147	–	–
diag 8	291,593	291,593	–	–	406,392	406,392	–	–
split 3	**333**	1929	664	3137	492	2096	811	3322
split 4	**1833**	34,579	7144	68,999	3847	31,827	12,527	82,939
split 5	**9388**	–	90,877	1,572,515	27,135	–	207,627	–
split 6	**45,566**	–	–	–	–	–	–	–
split 7	**211,828**	–	–	–	–	–	–	–
split 8	–	–	–	–	–	–	–	–
opt 3	**252**	619	350	621	372	639	399	655
opt 4	**1330**	5591	2591	5666	2305	6092	3268	5837
opt 5	**6550**	51,465	20,987	51,431	23,529	63,504	29,124	54,586
opt 6	**30,634**	494,997	178,954	474,498	–	–	272,734	541,533
opt 7	**137,788**	–	–	–	–	–	–	–
opt 8	**601,970**	–	–	–	–	–	–	–



(a) Execution Time (ms) **(b)** Number of Nodes

Fig. 7 Clock Refinement: BWITP vs. FWITP

BLN are consistent with the results presented in [18]. In these models, by using valuation interpolation, both execution times and the size of the state space can be significantly reduced. In particular, configuration BFB significantly outperforms all the other configurations.

In general, all configurations benefited greatly from the manual optimization that we applied for models opt n. However, using valuation interpolation still significantly improves performance for all configurations. Moreover, configuration BFB is still by far the most successful configuration. This also highlights the beneficial effects of combining abstraction refinement strategies for clock and discrete variables, in line with our results in [31].

In case of models diag n, clock refinement strategy LU is not applicable. The other four configurations, using FWITP for the handling of clocks, perform well regardless of search strategy, with BFN being the fastest. In fact, in case of this particular model, not eliminating diagonal constraints, and using zone interpolation seems to be the best of the examined approaches. Moreover, as Figure 7, shows, there is a significant difference in the performance of the two interpolation strategies, with FWITP having the better performance.

Finally, we point out that in case a model with diagonal constraints is analyzed by applying zone interpolation on its own (e.g. without zone splitting [5]), then termination is not guaranteed. In particular, during our experiments, we found that the algorithm diverges on the well-known example presented in [8].

6 Conclusions

In this paper, we presented *an algorithmic framework* for the lazy abstraction based location reachability checking of timed automata with discrete variables. We formalized in our framework several abstract domains and refinement strategies, both for clock and discrete variables, including a novel strategy that propagates valuation interpolants forward (called forward valuation interpolation). We also formalized the combination of abstractions and proved their properties. This framework allowed the straightforward implementation of efficient model checkers using configurable combined strategies.

We performed *empirical evaluation* on 16 configurations that our tool currently supports for 51 timed automata models, including ones that contain many discrete variables or diagonal

constraints. Our results show that our framework offers configurations that are competitive in terms of performance with the state-of-the-art. It turned out that for models with a significant number of discrete variables (category MCTA) it is worth using forward zone interpolation, combined with backward valuation interpolation. For the examined models with diagonal constraints, using forward interpolation for the handling of clocks performs well. In case of eager elimination of diagonal constraints by introducing new discrete variables, again a combination of strategies (breadth-first search with forward zone interpolation and backward valuation interpolation) is the most successful strategy. In general, using valuation interpolation in this model category, both execution times and the size of the state space can be significantly reduced.

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, octahedra, or polyhedra), and investigate alternative refinement strategies. Moreover, we plan to explore more sophisticated strategies for finding covering states, as this can potentially yield considerable speedups for our method.

Funding Open access funding provided by Budapest University of Technology and Economics.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, chap. 26, pp. 825–885. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-825
3. 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
4. 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
5. Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: ACPN 2003, LNCS, vol. 3098, pp. 87–124. Springer (2004). https://doi.org/10.1007/978-3-540-27755-2_3
6. Bérard, B., Petit, A., Diekert, V., Gastin, P.: Characterization of the expressive power of silent transitions in timed automata. Fundamenta Informaticae **36**(2,3), 145–182 (1998). https://doi.org/10.3233/FI-1998-36233
7. 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
8. Bouyer, P.: Untameable timed automata! In: STACS 2003, LNCS, vol. 2607, pp. 620–631. Springer (2003). https://doi.org/10.1007/3-540-36494-3_54
9. Bouyer, P., Chevalier, F.: On conciseness of extensions of timed automata. J. Autom. Lang. Combin. **10**(4), 393–405 (2005). https://doi.org/10.25596/jalc-2005-393

10. Bouyer, P., Laroussinie, F., Reynier, P.A.: Diagonal constraints in timed automata: forward analysis of timed systems. In: FORMATS 2005, LNCS, vol. 3829, pp. 112–126. Springer (2005). https://doi.org/10.1007/11603009_10
11. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI 2011, LNCS, vol. 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
12. Carioni, A., Ghilardi, S., Ranise, S.: MCMT in the land of parametrized timed automata. In: International Verification Workshop (VERIFY-2010), pp. 47–64 (2010)
13. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theories. In: TACAS 2008, LNCS, vol. 4963, pp. 397–412. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_30
14. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
15. 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
16. 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
17. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: CAV 1989, LNCS, vol. 407, pp. 197–212. Springer (1990). https://doi.org/10.1007/3-540-52148-8_17
18. Gastin, P., Mukherjee, S., Srivathsan, B.: Reachability in timed automata with diagonal constraints. In: International Conference on Concurrency Theory, LIPIcs, vol. 118, pp. 1–17. Dagstuhl (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.28
19. 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
20. 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. Dagstuhl (2011). https://doi.org/10.4230/LIPIcs.FSTTCS.2011.78
21. 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
22. 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
23. 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
24. 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
25. 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
26. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV 2003, LNCS, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
27. 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
28. Reynier, P.A.: Diagonal constraints handled efficiently in UPPAAL. Technical Report, LSV-07-02, Laboratoire Spécification et Vérification, ENS Cachan, France (2007)
29. 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
30. 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
31. Tóth, T., Majzik, I.: Lazy reachability checking for timed automata with discrete variables. In: SPIN 2018, LNCS, vol. 10869, pp. 235–254. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_14
32. Tóth, T., Majzik, I.: Supplementary material for the paper "Configurable verification of timed automata with discrete variables". Zenodo (2020). https://doi.org/10.5281/zenodo.3965792
33. 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