Abstraction Refinement Algorithms for Timed Automata*

Victor Roussanaly, Ocan Sankur, Nicolas Markey

Univ Rennes, Inria, CNRS, IRISA – Rennes, France

Abstract. We present abstraction-refinement algorithms for model checking safety properties of timed automata. The abstraction domain we consider abstracts away zones by restricting the set of clock constraints that can be used to define them, while the refinement procedure computes the set of constraints that must be taken into consideration in the abstraction so as to exclude a given spurious counterexample. We implement this idea in two ways: an enumerative algorithm where a lazy abstraction approach is adopted, meaning that possibly different abstract domains are assigned to each exploration node; and a symbolic algorithm where the abstract transition system is encoded with Boolean formulas.

1 Introduction

Model checking [26,10,12,4] is an automated technique for verifying that the set of behaviors of a computer system satisfies a given property. Model-checking algorithms explore finite-state automata (representing the system under study) in order to decide if the property holds; if not, the algorithm returns an explanation. These algorithms have been extended to verify real-time systems modelled as timed automata [3,2], an extension of finite automata with clock variables to measure and constrain the amount of time elapsed between occurrences of transitions. The state-space exploration can be done by representing clock constraints efficiently using convex polyhedra called *zones* [9,8]. Algorithms based on this data structure have been implemented in several tools such as Uppaal [7], and have been applied in various industrial cases.

The well-known issue in the applications of model checking is the *state-space explosion* problem: the size of the state space grows exponentially in the size of the description of the system. There are several sources for this explosion: the system might be made of the composition of several subsystems (such as a distributed system), it might contain several discrete variables (such as in a piece of software), or it might contain a number of real-valued clocks as in our case.

Numerous attempts have been made to circumvent this problem. Abstraction is a generic approach that consists in simplifying the model under study, so as to make it easier to verify [13]. *Existential* abstraction may only add extra behaviors, so that when a safety property holds in an abstracted model, it also

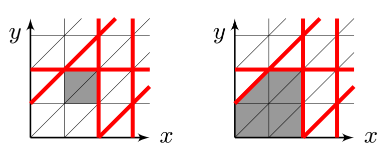
* This work was funded by ANR project Ticktac (ANR-18-CE40-0015) and by ERC grant EQualIS (StG-308087).

holds in the original model; if on the other hand a safety property fails to hold, the model-checking algorithms return a witness trace exhibiting the non-safe behaviour: this either invalidates the property on the original model, if the trace exists in that model, or gives information about how to automatically refine the abstraction. This approach, named CEGAR (counter-example guided abstraction refinement) [11], was further developed and used, for instance, in software verification (BLAST [20], SLAM [5], ...).

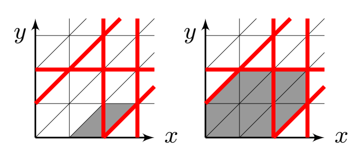
The CEGAR approach has been adapted to timed automata, e.g. in [14,18], but the abstractions considered there only consist in removing clocks and discrete variables, and adding them back during refinement. So for most well-designed models, one ends up adding all clocks and variables which renders the method useless. Two notable exceptions are [22], in which the zone extrapolation operators are dynamically adapted during the exploration, and [28], in which zones are refined when needed using interpolants. Both approaches define “exact” abstractions in the sense that they make sure that all traces discovered in the abstract model are feasible in the concrete model at any time.

In this work, we consider a more general setting and study *predicate abstractions* on clock variables. Just like in software model checking, we define abstract state spaces using these predicates, where the values of the clocks and their relations are approximately represented by these predicates. New predicates are generated if needed during the refinement step. We instantiate our approach by two algorithms. The first one is a zone-based enumerative algorithm inspired by the *lazy abstraction* in software model checking [19], where we assign a possibly different abstract domain to each node in the exploration. The second algorithm is based on binary decision diagrams (BDD): by exploiting the observation that a small number of predicates was often sufficient to prove safety properties, we use an efficient BDD encoding of zones similar to one introduced in early work [27].

Let us explain the abstract domains we consider. Assume there are two clock variables x and y. The abstraction we consider consists in restricting the clock constraints that can be used when defining zones. Assume that we only allow to compare x with 2 or 3; that y can only be compared with 2, and $x-y$ can only be compared with -1 or 2. Then any conjunction of constraints one might obtain in this manner will be delimited by the thick red lines in Fig. 1; one cannot define a finer region under this restriction. The figure shows the abstraction process: given



(a) Abstraction of zone $1 \le x, y \le 2$



(b) Abstraction of zone $y \le 1 \wedge 1 \le x - y \le 2$

Fig. 1: The abstract domain is defined by the clock constraints shown in thick red lines. In each example, the abstraction of the zone shown on the left (shaded area) is the larger zone on the right.

a “concrete” zone, its abstraction is the smallest zone which is a superset and is definable under our restriction. For instance, the abstraction of $1 \leq x, y \leq 2$ is $0 \leq x, y \leq 2 \wedge -1 \leq x - y$ (cf. Fig. 1a).

Related Works. We give more detail on zone abstractions in timed automata. Most efforts in the literature have been concentrated in designing zone abstraction operators that are exact in the sense that they preserve the reachability relation between the locations of a timed automaton; see [6]. The idea is to determine bounds on the constants to which a given clock can be compared to in a given part of the automaton, since the clock values do not matter outside these bounds. In [21,22], the authors give an algorithm where these bounds are dynamically adapted during the exploration, which allows one to obtain coarser abstractions. In [28], the exploration tree contains pairs of zones: a concrete zone as in the usual algorithm, and a coarser abstract zone. The algorithm explores all branches using the coarser zone and immediately refines the abstract zone whenever an edge which is disabled in the concrete zone is enabled. In [17], a CEGAR loop was used to solve timed games by analyzing strategies computed for each abstract game. The abstraction consisted in collapsing locations.

Some works have adapted the abstraction-refinement paradigm to timed automata. In [14], the authors apply “localization reduction” to timed automata within an abstraction-refinement loop: they abstract away clocks and discrete variables, and only introduce them as they are needed to rule out spurious counterexamples. A more general but similar approach was developed in [18]. In [30], the authors adapt the trace abstraction refinement idea to timed automata where a finite automaton is maintained to rule out infeasible edge sequences.

The CEGAR approach was also used recently in the LinAIG framework for verifying linear hybrid automata [1]. In this work, the backward reachability algorithm exploits *don't-cares* to reduce the size of the Boolean circuits representing the state space. The abstractions consist in enlarging the size of *don't-cares* to reduce the number of linear predicates used in the representation.

2 Timed Automata and Zones

2.1 Timed automata

Given a finite set of clocks $\mathcal{C}$, we call *valuations* the elements of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$. For a clock valuation v, a subset $R \subseteq \mathcal{C}$, and a non-negative real d, we denote with $v[R \leftarrow d]$ the valuation w such that $w(x) = v(x)$ for $x \in \mathcal{C} \setminus R$ and $w(x) = d$ for $x \in R$, and with $v + d$ the valuation w' such that $w'(x) = v(x) + d$ for all $x \in \mathcal{C}$. We extend these operations to sets of valuations in the obvious way. We write $\mathbf{0}$ for the valuation that assigns 0 to every clock. An *atomic guard* is a formula of the form $x \prec k$ or $x - y \prec k$ with $x, y \in \mathcal{C}$, $k \in \mathbb{N}$, and $\prec \in \{<, \leq, >, \geq\}$. A *guard* is a conjunction of atomic guards. A valuation v satisfies a guard g, denoted $v \models g$, if all atomic guards hold true when each $x \in \mathcal{C}$ is replaced with $v(x)$. Let $[\![g]\!] = \{v \in \mathbb{R}_{\geq 0}^{\mathcal{C}} \mid v \models g\}$ denote the set of valuations satisfying g. We write $\Phi_{\mathcal{C}}$ for the set of guards built on $\mathcal{C}$.

A *timed automaton* $\mathcal{A}$ is a tuple $(\mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$, where $\mathcal{L}$ is a finite set of locations, $\mathsf{Inv}\colon \mathcal{L} \to \Phi_{\mathcal{C}}$ defines location invariants, $\mathcal{C}$ is a finite set of clocks, $E \subseteq \mathcal{L} \times \Phi_{\mathcal{C}} \times 2^{\mathcal{C}} \times \mathcal{L}$ is a set of edges, and $\ell_0 \in \mathcal{L}$ is the initial location. An edge $e = (\ell, g, R, \ell')$ is also written as $\ell \xrightarrow{g,R} \ell'$. For any location ℓ, we let $E(\ell)$ denote the set of edges leaving ℓ.

A *configuration* of $\mathcal{A}$ is a pair $q = (\ell, v) \in \mathcal{L} \times \mathbb{R}^{\mathcal{C}}_{\geq 0}$ such that $v \models \mathsf{Inv}(\ell)$. A *run* of $\mathcal{A}$ is a sequence $q_1 e_1 q_2 e_2 \ldots q_n$ where for all $i \geq 1$, $q_i = (\ell_i, v_i)$ is a configuration, and either $e_i \in \mathbb{R}_{>0}$, in which case $q_{i+1} = (\ell_i, v_i + e_i)$, or $e_i = (\ell_i, g_i, R_i, \ell_{i+1}) \in E$, in which case $v_i \models g_i$ and $q_{i+1} = (\ell_{i+1}, v_i[R_i \leftarrow 0])$. A *path* is a sequence of edges with matching endpoint locations.

2.2 Zones and DBMs

Several tools for timed automata implement algorithms based on *zones*, which are particular polyhedra definable with clock constraints. Formally, a zone Z is a subset of $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ definable by a guard in $\Phi_{\mathcal{C}}$.

We recall a few basic operations defined on zones. First, the intersection $Z \cap Z'$ of two zones Z and Z' is clearly a zone. Given a zone Z, the set of time-successors of Z, defined as $Z{\uparrow} = \{v + t \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid t \in \mathbb{R}_{\geq 0},\ v \in Z\}$, is easily seen to be a zone; similarly for time-predecessors $Z{\downarrow} = \{v \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid \exists t \geq 0.\ v + t \in Z\}$. Given $R \subseteq \mathcal{C}$, we let $\mathsf{Reset}_R(Z)$ be the zone $\{v[R \leftarrow 0] \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid v \in Z\}$, and $\mathsf{Free}_x(Z) = \{v' \in \mathbb{R}^{\mathcal{C}}_{\geq 0} \mid \exists v \in Z, d \in \mathbb{R}_{\geq 0}, v' = v[x \leftarrow d]\}$.

Zones can be represented as *difference-bound matrices (DBM)* [15,8]. Let $\mathcal{C}_0 = \mathcal{C} \cup \{0\}$, where 0 is an extra symbol representing a special clock variable whose value is always 0. A DBM is a $|\mathcal{C}_0| \times |\mathcal{C}_0|$-matrix taking values in $(\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. Intuitively, cell (x, y) of a DBM M stores a pair $(d, \prec)$ representing an upper bound on the difference $x - y$. For any DBM M, we let $[\![M]\!]$ denote the zone it defines.

While several DBMs can represent the same zone, each zone admits a *canonical* representation, which is obtained by storing the tightest clock constraints defining the zone. This canonical representation can be obtained by computing shortest paths in a graph where the vertices are clocks and the edges weighted by clock constraints, with natural addition and comparison of elements of $(\mathbb{Z} \times \{<, \leq\}) \cup \{(+\infty, <)\}$. This graph has a negative cycle if, and only if, the associated DBM represents the empty zone.

All the operations on zones can be performed efficiently (in $O(|\mathcal{C}_0|^3)$) on their associated DBMs while maintaining reduced form. For instance, the intersection $N = Z \cap Z'$ of two canonical DBMs Z and Z' can be obtained by first computing the DBM $M = \min(Z, Z')$ such that $M(x, y) = \min\{Z(x, y), Z'(x, y)\}$ for all $(x, y) \in {\mathcal{C}_0}^2$, and then turning M into canonical form. We refer to [8] for full details. By a slight abuse of notation, we use the same notations for DBMs as for zones, writing e.g. $M' = M{\uparrow}$, where M and M' are reduced DBMs such that $[\![M']\!] = [\![M]\!]{\uparrow}$. Given an edge $e = (\ell, g, R, \ell')$, and a zone Z, we define $\mathsf{Post}_e(Z) = \mathsf{Inv}(\ell') \cap (g \cap \mathsf{Reset}_R(Z)){\uparrow}$, and $\mathsf{Pre}_e(Z) = (g \cap \mathsf{Free}_R(\mathsf{Inv}(\ell') \cap Z)){\downarrow}$.

For a path $\rho = e_1 e_2 \dots e_n$, we define Post_ρ and Pre_ρ by iteratively applying Post_{e_i} and Pre_{e_i} respectively.

2.3 Clock-predicate abstraction and interpolation

For all clocks x and y in $\mathcal{C}_0$, we consider a finite set $\mathcal{D}_{x,y} \subseteq \mathbb{N} \times \{\leq, <\}$, and gather these in a table $\mathcal{D} = (\mathcal{D}_{x,y})_{x,y \in \mathcal{C}_0}$. $\mathcal{D}$ is the *abstract domain* which restricts zones to be defined only using constraints of the form $x - y \prec k$ with $(k, \prec) \in \mathcal{D}_{x,y}$, as seen earlier. Let us call $\mathcal{D}$ the *concrete domain* if $\mathcal{D}_{x,y} = \mathbb{N} \times \{\leq, <\}$ for all $x, y \in \mathcal{C}_0$. A zone Z is $\mathcal{D}$-definable if there exists a DBM D such that $Z = [\![D]\!]$ and $D(x,y) \in \mathcal{D}_{x,y}$ for all $x, y \in \mathcal{C}_0$. Note that we do not require this witness DBM D to be reduced; the reduction of such a DBM might introduce additional values. We say that domain $\mathcal{D}'$ is a *refinement* of $\mathcal{D}$ if for all $x, y \in \mathcal{C}_0$, we have $\mathcal{D}_{x,y} \subseteq \mathcal{D}'_{x,y}$.

An abstract domain $\mathcal{D}$ induces an *abstraction function* $\alpha_{\mathcal{D}} \colon 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}} \to 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}}$ where $\alpha_{\mathcal{D}}(Z)$ is the smallest $\mathcal{D}$-definable zone containing Z. For any reduced DBM D, $\alpha_{\mathcal{D}}([\![D]\!])$ can be computed by setting $D'(x,y) = \min\{(k, \prec) \in \mathcal{D}_{x,y} \mid D(x,y) \leq (k, \prec)\}$ (with $\min \emptyset = (\infty, <)$).

An *interpolant* for a pair of zones (Z_1, Z_2) with $Z_1 \cap Z_2 = \emptyset$ is a zone Z_3 with $Z_1 \subseteq Z_3$ and $Z_3 \cap Z_2 = \emptyset$[1] [28]. We use interpolants to refine our abstractions; in order not to add too many new constraints when refining, our aim is to find *minimal interpolants*: define the density of a DBM D as $d(D) = \#\{(x,y) \in {\mathcal{C}_0}^2 \mid D(x,y) \neq (\infty, <)\}$. Notice that while any pair of disjoint convex polyhedra can be separated by hyperplanes, not all pairs of disjoint zones admit interpolants of density 1; this is because not all (half-spaces delimited by) hyperplanes are zones.

Lemma 1. *There exist pairs of zones accepting no simple interpolants.*

Proof. Consider 3-dimensional zones A, defined as $z = 0 \wedge x = y$, and B, defined as $y \geq 2 \wedge z \leq 2 \wedge y - x \leq 1 \wedge x - z \leq 1$. Both zones and their canonical DBMs are represented on Fig. 2.

We observe that they are disjoint: if a triple (x, y, z) were in both A and B, then $x = y$ and $z = 0$ (for being in B); in A, $y \geq 2$, hence also $x \geq 2$, contradicting $x - z \leq 1$.

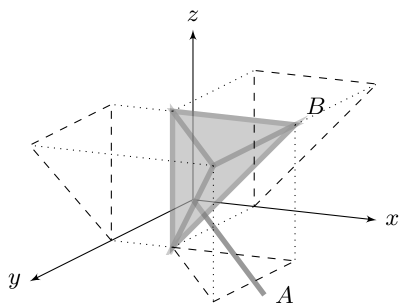
Now, assume that there is a simple interpolant I, with $A \cap I = \emptyset$ and $B \subseteq I$. In the canonical DBM of I, only one non-diagonal element is not $(+\infty, <)$; assume $I(x,y) \neq (+\infty, <)$. Then we must have $A(y,x) + I(x,y) < (0, \leq)$, and $B(x,y) \leq I(x,y)$. Then $A(x,y) + B(x,y) < (0, \leq)$. However, it can be observed that in our example, $A(x,y) + B(y,x) \geq (0, \leq)$ for all pairs (x,y). $\square$

Still, we can bound the density of a minimal interpolant:

Lemma 2. *For any pair of disjoint, non-empty zones (A, B), there exists an interpolant of density less than or equal to $|\mathcal{C}_0|/2$.*

---

[1] It is sometimes also required that the interpolant only involves clocks that have non-trivial constraints in both Z_1 and Z_2. We do not impose this requirement in our definition, but it will hold true in the interpolants computed by our algorithm.



$$A = \begin{pmatrix} (0,\le) & (+\infty,<) & (+\infty,<) & (0,\le) \\ (+\infty,<) & (0,\le) & (0,\le) & (+\infty,<) \\ (+\infty,<) & (0,\le) & (0,\le) & (+\infty,<) \\ (0,\le) & (+\infty,<) & (+\infty,<) & (0,\le) \end{pmatrix}$$

$$B = \begin{pmatrix} (0,\le) & (3,\le) & (4,\le) & (2,\le) \\ (-1,\le) & (0,\le) & (1,\le) & (1,\le) \\ (-2,\le) & (1,\le) & (0,\le) & (0,\le) \\ (0,\le) & (1,\le) & (2,\le) & (0,\le) \end{pmatrix}$$

Fig. 2: Two zones that cannot be separated by a simple interpolant

Proof. Assume that A and B are given as canonical DBMs, which we also write A and B for the sake of readability. We prove the stronger result that $A \cap B = \emptyset$ if, and only if, for some $n \le |\mathcal{C}_0|/2$, there exists a sequence of pairwise-distinct clocks $(x_i)_{0\le i\le 2n-1}$ such that, writing $x_{2n} = x_0$,

$$\sum_{i=0}^{n-1} A(x_{2i}, x_{2i+1}) + B(x_{2i+1}, x_{2i+2}) < (0, \le).$$

Before proving this result, we explain how we conclude the proof: the inequality above entails that

$$\bigcap_{i=0}^{n-1} A(x_{2i}, x_{2i+1}) \cap \bigcap_{i=0}^{n-1} B(x_{2i+1}, x_{2i+2}) = \emptyset$$

where we abusively identify $A(x, y)$ with the half-space it represents. It follows that $\bigcap_{i=0}^{n-1} B(x_{2i+1}, x_{2i+2})$ is an interpolant, whose density is less than or equal to $|\mathcal{C}_0|/2$.

Assume that such a sequence exists, and write C for the DBM $\min(A, B)$. Then

$$\begin{aligned} \sum_{i=0}^{2n-1} C(x_i, x_{i+1}) &= \sum_{i=0}^{2n-1} \min\{A(x_i, x_{i+1}), B(x_i, x_{i+1})\} \\ &\le \sum_{i=0}^{n-1} A(x_{2i}, x_{2i+1}) + B(x_{2i+1}, x_{2i+2}) < (0, \le). \end{aligned}$$

This entails that the intersection is empty.

Conversely, if the intersection is empty, then there is a sequence of clocks $(x_i)_{0\le i<m}$, with $m \le |\mathcal{C}_0|$, such that, letting $x_m = x_0$, we have

$$\sum_{i=0}^{m-1} \min\{A(x_i, x_{i+1}), B(x_i, x_{i+1})\} < (0, \le).$$

Algorithm 1: Algorithm for minimal interpolant

Input: canonical DBM A, B

;
for $(x, y) \in {\mathcal{C}_0}^2$ **do**
  if $A(x, y) \leq B(x, y)$ **then**
    $M^0(x, y) := A(x, y)$;
  else
    $M^0(x, y) := (\infty, <)$;
$N^0 := \mathsf{canonical}(M^0)$;
for *(*$i = 1$*;* $i \leq |\mathcal{C}_0|/2$*;* $i++$*)* **do**
  for $(x, y) \in {\mathcal{C}_0}^2$ **do**
    $M^i(x, y) := \min\{N^{i-1}(x, y), \min_{z \in \mathsf{SC}_B(y)} N^{i-1}(x, z) + B(z, y)\}$;
  for $(x, y) \in {\mathcal{C}_0}^2$ **do**
    $N^i(x, y) := \min\{M^i(x, y), \min_{z \in \mathsf{SC}_A(y)} M^i(x, z) + A(z, y)\}$;
    if $(x = y)$ *and* $N^i(x, x) < (0, \leq)$ **then**
      return $(\texttt{true}, i)$;
return `false`;

Consider one of the shortest such sequences. Since A and B are non-empty, the sum must involve at least one element of each DBM. Moreover, if it involves two consecutive elements of the same DBM (i.e., if $A(x_i, x_{i+1}) < B(x_i, x_{i+1})$ and $A(x_{i+1}, x_{i+2}) < B(x_{i+1}, x_{i+2})$ for some i), then by canonicity of the DBMs of A and B, we can drop clock x_{i+1} from the sequence and get a shorter sequence satisfying the same inequality, contradicting minimality of our sequence. The result follows. □

By adapting the algorithm of [28] for computing interpolants, we can compute minimal interpolants efficiently:

Proposition 3. *Computing a minimal interpolant can be performed in* $O(|\mathcal{C}|^4)$.

Proof. Algorithm 1 describes our procedure. In order to prove its correctness, we begin with proving that the sequence of DBM it computes satisfies the following property:

Lemma 4. *For any* $i \geq 0$ *such that* N^i *has been computed by Algorithm 1, for any* $(x, y) \in {\mathcal{C}_0}^2$*, it holds*

$$N^i(x, y) = \min_{\substack{\pi \in \mathsf{Paths}(x,y) \\ |\pi|_B \leq i}} W_{\min(A,B)}(\pi).$$

Proof. The proof proceeds by induction on i. For $i = 0$, pick a path $\pi = (x_i)_{0\leq i\leq k}$ from $x+0$ to x_k such that $|\pi|_B = 0$. Then

$$\begin{aligned} W_{\min(A,B)}(\pi) &= \sum_{0\leq i<k} A(x_i, x_{i+1}) = \sum_{0\leq i<k} M^0(x_i, x_{i+1}) \\ &= \sum_{0\leq i<k} N^0(x_i, x_{i+1}) \geq N^0(x_0, x_k). \end{aligned}$$

The first two equalities follow from the fact that π only involves transitions in $E_{A\leq B}$; the third equality is because canonization will not modify entries from A (since A is originally in canonical form). The last inequality follows from canonicity of N^0.

Now assume that the result holds at step i, and that N^{i+1} is defined. Pick x and y in $\mathcal{C}_0$. By construction of M^{i+1} and $N^{i+!}$, there exists z and t in $\mathcal{C}_0$ such that $N^{i+1}(x,y) = N^i(x,z) + B(z,t) + A(t,y)$ with $t \in \mathsf{SC}_A(y)$ (or $t = y$) and $z \in \mathsf{SC}_B(t)$ (or $z = t$). From the induction hypothesis, there is a path π' from x to z such that $N^i(x,z) = W_{\min(A,B)}(\pi')$, and $|\pi'|_B \leq i$. Adding t and y to this path, we get a path π from x to y such that $N^{i+1}(x,y) = W_{\min(A,B)}(\pi)$ and $|\pi|_B \leq i+1$.

It remains to prove that any path π from x to y with $|\pi|_B \leq i+1$ is such that $N^{i+1}(x,y) \leq W_{\min(A,B)}(\pi)$. Fix such a path $\pi = (x_j)_{0\leq j\leq k}$ we concentrate on the case where $|\pi|_B = i+1$, since the other case follows from the induction hypothesis. We decompose π as $\pi_1 = (x_j)_{0\leq j\leq l}$, $\pi_2 = (x_l, x_{l+1})$ and $\pi_3 = (x_j)_{l+1\leq j\leq k}$, such that $(x_l, x_{l+1}) \in E_B$ and $(x_j, x_{j+1}) \in E_A$ for all $j > l$; in other terms, π_2 is the last E_B transition of π, and π_3 is a path from x_{l+1} to x_k only involving transitions in A. Then

$$w_{\min(A,B)}(\pi_2 \cdot \pi_3) = B(x_l, x_{l+1}) + w_A(\pi_3) \geq B(x_l, x_{l+1}) + w_A(x_{l+1}, x_k),$$

and

- if $(x_{l+1}, x_k) \in E_B$, then $w_{\min(A,B)}(\pi_2 \cdot \pi_3) \geq w_B(x_l, x_k)$, and, applying the induction hypothesis, $w_{\min(A,B)}(\pi) \geq N^i(x, x_l) + \min\{A(x_l, x_k), B(x_l, x_k)\}$. Since $M^{i+1}(x, x_l) \leq N^i(x, x_l)$, we get

$$\begin{aligned} w_{\min(A,B)}(\pi) &\geq \min\{N^i(x, x_l) + B(x_l, x_k), M^{i+1}(x, x_l) + A(x_l, x_k)\} \\ &\geq N^{i+1}(x, x_k). \end{aligned}$$

- if $(x_{l+1}, x_k) \in E_B$, then $w_{\min(A,B)}(\pi_2 \cdot \pi_3) \geq B(x_l, x_{l+1}) + A(x_{l+1}, x_k)$. From the induction hypothesis, $w_{\min(A,B)}(\pi) \geq N^i(x, x_l) + B(x_l, x_{l+1}) + A(x_{l+1}, x_k) \geq N^{i+1}(x, x_k)$. □

Following the argument of the proof of Lemma 2, we get:

Corollary 5. *If $A \cap B \neq \emptyset$, then Algorithm 1 returns* `false`*; otherwise, it returns* $(\texttt{true}, k)$ *for the smallest k such that for all cyclic path π such that $|\pi|_B < k$, it holds $w_{\min(A,B)}(\pi) \geq (0, \leq)$.*

This entails that k is the dimension of the minimal interpolant. The minimal interpolant can be obtained by taking the B-elements of the negative cycle found by the algorithm. □

3 Enumerative Algorithm

The first type of algorithm we present is a zone-based enumerative algorithm based on the clock-predicate abstractions. Let us first describe the overall algorithm in Algorithm 2, which is a typical abstraction-refinement loop. We then explain how the abstract reachability and refinement procedures are instantiated.

Algorithm 2: Enumerative algorithm checking the reachability of a target location ℓ_T.

Input: $\mathcal{A} = (\mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$, ℓ_T

1 Initialize $\mathcal{D}_0$;
2 wait:= $\{\mathsf{node}(\ell_0, \mathbf{0}\uparrow, \mathcal{D}_0)\}$;
3 passed:= $\emptyset$;
4 **while do**
5     $\pi :=$ AbsReach($\mathcal{A}$, wait, passed, ℓ_T);
6   **if** $\pi = \emptyset$ **then**
7   **return** Not reachable;
8   **else**
9   **if** *trace* π *is feasible* **then**
10   **return** Reachable;
11   **else**
               Refine(π, wait, passed);
12 **return** Not reachable;

Algorithm 3: AbsReach

Input: $(\mathcal{L}, \mathsf{Inv}, l_0, \mathcal{C}, E)$, wait, passed, ℓ_T

1 **while** wait $\neq \emptyset$ **do**
2     $n :=$ wait.$pop()$;
3   **if** $n.\ell = \ell_T$ **then**
4   **return** Trace from root to n;
5   **if** $\exists n' \in$ passed *such that* $n.\ell = n'.\ell \wedge n.Z \subseteq n'.Z$ **then**
6   n.covered $:= n'$;
7   **else**
8         $n.Z := \alpha(n.Z, n)$;
9   passed.$add(n)$;
10   **for** $e = (\ell, g, R, \ell') \in E(n.\ell)$ *s.t.* $Z' := \mathrm{Post}_e(n.Z) \neq \emptyset$ **do**
11             $\mathcal{D}' :=$ choose-dom(n, e);
12             $n' :=$ node$(\ell', Z', \mathcal{D}')$;
13   n'.parent $:= n$;
14   wait.$add(n')$;
15 **return** $\emptyset$;

The initialization at line 1 chooses an abstract domain for the initial state, which can be either empty (thus the coarsest abstraction) or defined according to some heuristics. The algorithm maintains the wait and passed lists that are used in the forward exploration. As usual, the wait list can be implemented as a stack, a queue, or another priority list that determines the search order. The algorithm also uses covering nodes. Indeed if there are two node n and n', with $n \in$ passed, $n' \in$ wait, $n.\ell = n'.\ell$, and $n'.z \subseteq n.Z$, then we know that every location reachable from n' is also reachable from n. Since we have already explored n and we generated its successors, there is no need to explore the successors of n'. The algorithm explicitly creates an exploration tree: line 2 creates a node containing location ℓ_0, zone $\mathbf{0}\uparrow$, and the abstract domain $\mathcal{D}_0$ as the root of our tree, and adds this to the wait list. More details on the tree are given in the next subsection. Procedure AbsReach then looks for a trace to the target location ℓ_T. If such a trace exists, line 9 checks its feasibility. Here π is a sequence of node and edges of $\mathcal{A}$. The feasibility check is done by computing predecessors with zones starting from the final state, without using the abstraction function. If the last zone intersects our initial zone, this means that the trace is feasible. More details are given in Section 3.2.

3.1 Abstract forward reachability: AbsReach

We give a generic algorithm independently from the implementations of the abstraction functions and the refinement procedure.

Algorithm 3 describes the reachability procedure under a given abstract domain $\mathcal{D}$. It is similar to the standard forward reachability algorithm using a wait-list and a passed-list. We explicitly create an exploration tree where the leaves are nodes in wait, covered nodes, or nodes that have no non-empty successors. Each node n contains the fields ℓ, Z which are labels describing the current location and zone; field covered points to a node covering the current node (it is undefined if the current node is not (known to be) covered); field parent points to the parent node in the tree (it is undefined for the root); and field $\mathcal{D}$ is the abstract domain associated with the node. Thus, the algorithm uses a possibly different abstract domain for each node in the exploration tree.

The difference of our algorithm w.r.t. the standard reachability can be seen at lines 8 and 11. At line 8, we apply the abstraction function to the zone taken from the wait-list before adding it to the passed-list. The abstraction function α is a function of a zone Z and a node n. This allows one to define variants with different dependencies; for instance, α might depend on the abstract domain $n.\mathcal{D}$ at the current node, but it can also use other information available in n or on the path ending in n. For now, it is best to think of α simply as $Z \mapsto \alpha_{n.\mathcal{D}}(Z)$. At line 11, the function choose-dom chooses an abstract domain for the node n'.

In our implementation, the abstraction function always abstracts the given zone w.r.t. the abstract domain $n.\mathcal{D}$. For choose-dom, we considered three variants:

- one using a global domain, the same for all nodes: this way, each refinement benefits to all nodes, but this is often a drawback since in general different parts of the automaton will better have different abstract domain;
- one using a local domain for each node: this has the advantage of using the coarsest possible abstraction, but it takes more memory and usually involves more refinements.
- one using one domain per location of the automaton: this appears to be a good trade-off between the above two approaches.

Remark 1. Note that we use the abstraction function when the node is inserted in the passed list. This is because we want the node to contain the smallest zone possible when we test whether the node is covered. We only need to use the abstracted zone when we compute its successor and when we test whether the node is covering. This allows us to store a unique zone.

As a first step towards proving correctness of our algorithm, we consider that the following property is preserved by Algorithm AbsReach:

For all nodes n in passed, for all edges e from $n.\ell$, if $\mathsf{Post}_e(n.Z) \neq \emptyset$, then n has a child n' such that $\mathsf{Post}_e(n.Z) \subseteq n'.Z$. If n' is in passed, then we also have $\alpha_{n'.\mathcal{D}}(\mathsf{Post}_e(n.Z)) \subseteq n'.Z$. (1)

Algorithm 4: Refine

Input: π, wait, passed

;
$n :=$ last node of π;
$Z := n.Z$;
$r :=$ Refine-rec$(n, Z,$ wait, passed$)$;
$n_{cut} :=$ node to cut (according to heuristics);
cut(n_{cut});
if $n_{cut}.Z = \emptyset$ **then**
  delete n_{cut}
else
  passed.*remove*(n_{cut});
  wait.*add*(n_{cut});
  $n_{cut}.Z :=$ Concrete(n_{cut});
return r;

Algorithm 5: Refine-rec

Input: $n, Z,$ wait, passed

;
$C :=$ Concrete(n);
if $C \cap Z = \emptyset$ **then**
  Strengthen $(n, Z, C,$ wait$)$;
  return Not Feasible;
else if *n has no parent* **then**
  return Feasible ;
else
  e edge from n.parent to n;
  $Z' :=$ Pre$_e(Z) \cap n$.parent.Z;
  if Refine-rec$(n$.parent, Z', wait, passed$) =$ Feasible **then**
    return Feasible;
  else
    $C :=$Concrete(n);
    Strengthen$(n, Z, C,$ wait$)$;
    return Not Feasible;

Algorithm 6: Concrete

Input: n

;
if *n has parent* **then**
  $e :=$ edge from n.parent to n;
  return Post$_e(n$.parent.$Z)$;
else
  return initial zone;

Algorithm 7: Strengthen

Input: $n, Z, C,$ wait

;
if $\alpha_{n.\mathcal{D}}(C) \cap Z \neq \emptyset$ **then**
  $I :=$interpolant(C, Z);
  $n.\mathcal{D}.add(I)$;
$n := \alpha_{n.\mathcal{D}}(C)$;
Add every uncovered nodes to wait;

The following is an easy observation about our algorithm:

Lemma 6. *Algorithm* AbsReach *preserves Property* (1).

Note that although we use inclusion in Property (1), AbsReach would actually preserve equality of zones, but we will not always have equality before running AbsReach. This is because Refine might change the zones of some nodes without updating the zones of all their descendants.

3.2 Refinement: Refine

We now describe our refinement proecdure Refine. Let us now assume that AbsReach returns $\pi = A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{k-1}} A_k$, and write $\mathcal{D}_i$ for the domain associated with each A_i. We write C_1 for the initial concrete zone, and for $i < k$, we define $C_{i+1} = \mathsf{Post}_{\sigma_i}(A_i)$. We also note $Z_k = A_k$ and for $i < k$, $Z_i = \mathsf{Pre}_{\sigma_i}(Z_{i+1}) \cap A_i$. Then π is not feasible if, and only if, $\mathsf{Post}_{\sigma_1 \ldots \sigma_k}(C_1) = \emptyset$, or equivalently $\mathsf{Pre}_{\sigma_1 \ldots \sigma_k}(A_k) \cap C_1 = \emptyset$. Since for all $i < k$, it holds $C_i \subseteq A_{i+1}$, we have that π is not feasible if, and only if, $\exists i \leq k.\ C_i \cap Z_i = \emptyset$. We illustrate this on Fig. 3.

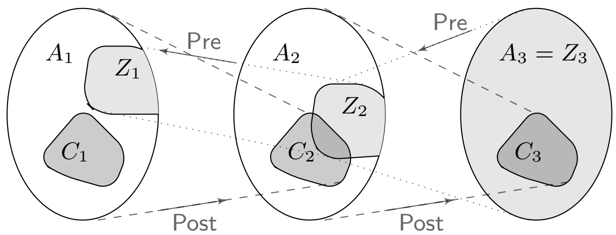


Fig. 3: Spurious counter-example: $Z_1 \cap C_1 = \emptyset$

Let us assume that π is not feasible. Let us denote by i_0 the maximal index such that $C_{i_0} \cap Z_{i_0} = \emptyset$. This index also has the property that for all $j < i_0$, we have $Z_j = \emptyset$ and $Z_{i_0} \neq \emptyset$. Once we have identified this trace as spurious by computing the Z_j, we have two possibilities:

– if $Z_{i_0} \cap \alpha_{\mathcal{D}_{i_0}}(C_{i_0}) \neq \emptyset$: this means that we can reach A_k from $\alpha_{\mathcal{D}_{i_0}}(C_{i_0})$ but not from C_{i_0}. In other words, our abstraction is too coarse and we must add some values to $\mathcal{D}_{i_0}$ so that $Z_{i_0} \cap \alpha_{\mathcal{D}_{i_0}}(C_{i_0}) = \emptyset$. Those values are found by computing the interpolant of Z_{i_0} and C_{i_0}
– Otherwise it means that $\alpha_{\mathcal{D}_{i_0}}(C_{i_0})$ cannot reach A_k and the only reason the trace exists is because either $\mathcal{D}_{i_0}$ or A_{i_0-1} has been modified at some point and A_{i_0} was not modified accordingly.

We can then update the values of C_i for $i > i_0$ and repeat the process until we reach an index j_0 such that $C_{j_0} = \emptyset$. We then have modified the nodes $n_{i_0}, \ldots, n_{j_0}$ and knowing that $n_{j_0}.Z = \emptyset$, we can delete it and all of its descendants. Since some of the descendants of n_{i_0} have not been modified, this might cause some refinements of the first type in the future. In order to ensure termination, we sometimes have to cut a subtree from a node in $n_{i_0}, \ldots, n_{j_0-1}$ and reinsert it in the wait list to restart the exploration from there. We call this action cut, and we can use several heuristics to decide when to use it. In the rest of this paper we will use the following heuristics: we perform cut on the first node of $n_{i_0} \ldots n_{j_0}$ that is covered by some other node. Since this node is covered, we know that we will not restart the exploration from this node, or that the node was covered by one of its descendant. If none of these nodes are covered, we delete n_{j_0} and its descendants. Other heuristics are possible, for instance applying cut on n_{i_0}. We found that the above heuristics was the most efficient in our experiments.

Lemma 7. *Pick a node n, and let $Y = n.Z$. Then after running* Refine*, either node n is deleted, or it holds $n.Z \subseteq Y$. In other words, the zone of a node can only be reduced by* Refine*.*

Proof. Refine may only add values to the domain of a node, so that the refined zone is included in the previous one. If no values were added to the domain of a node, then its parent must have been modified. Since $A \subseteq B \Rightarrow \mathsf{Post}_e(A) \subseteq \mathsf{Post}_e(B)$, the result follows by induction. $\square$

It follows that Refine also preserves Property (1), so that:

Lemma 8. *Algorithm 2 satisfies Property* (1).

Proof. We prove that procedure Refine preserves property (1). Combined with Lemma 6, this entails the result. First notice that Refine may not add new nodes. Let n' be a node, n its parent, and e the edge from n to n'. Three cases may arise:

- n' has been modified: then it must have been modified at line 4 or 15. At this point n is no longer modified, and Refine ensures that $\mathsf{Concrete}(n) \subseteq n'.Z$, and $\mathsf{Concrete}(n) = \mathsf{Post}_e(n.Z)$. If n' is in passed, we also have $\alpha_{n'.\mathcal{D}}(\mathsf{Post}_e(n.Z)) = n'.Z$;
- n' has been deleted: in this case, if n is part of the subtree and it has either been deleted or moved to wait, and is not in passed anymore. Otherwise n is not part of the subtree that has been cut and n' is the root of this subtree, with $n'.Z = \emptyset$ and since $\mathsf{Concrete}(n) \subseteq n'.Z$;
- n' has not been modified, but n has: then using Lemma 7, we know that the inclusion $\mathsf{Post}_e(n.Z) \subseteq n'.Z$ (or $\alpha_{n'.\mathcal{D}}(\mathsf{Post}_e(n.Z)) \subseteq n'.Z$) is preserved by Refine. $\square$

We can then prove that our algorithm correctly decides the reachability problem and always terminates.

Theorem 9. *Algorithm 2 terminates and is correct.*

Proof. We first prove correctness, assuming termination. First let us notice that if the wait set is empty, then for any reachable location l, there is a node n such that $n.\ell = l$. This is because we over-approximate the zones as shown in Lemma 8, so we over-approximate the set of reachable states. Thus, if $\mathsf{AbsReach}(\mathcal{A}, \mathsf{wait}, \mathsf{passed}, \ell_T)$ returns $\emptyset$, then ℓ_T is not reachable in $\mathcal{A}$. In other words, if the enumerative algorithm returns "*Not reachable*" then ℓ_T is indeed not reachable.

On the other hand, if the algorithm returns "*Reachable*", it means that there is a feasible trace reaching ℓ_T, and ℓ_T is indeed reachable.

We now prove termination. Since there are a finite number of possible locations and we can limit the number of possible zones to a finite number using abstraction functions, we can deduce that $\mathsf{AbsReach}$ terminates.

Let us assume that the enumerative algorithm does not terminate. Then it means that Refine is called infinitely many timed. Note that Refine is modifying a node and a node can be modified a finite number of time.

We can also note that a node can be destroyed only if one of its ancestors is modified. As such, we can show that for every depth k, there is a point in the algorithm where every node at depth k or less is fixed and will no longer be modified.

So we know that the algorithm does not terminate if, and only if, the depth of the resulting tree is unbounded. This means that there exists a path where we have two distinct nodes n_1 and n_2 with $n_1.\ell = n_2.\ell$ and $n_1.Z = n_2.Z$, since the number of location and possible zones is finite. Without loss of generality, we can assume that n_1 is an ancestor of n_2 and n_2 is the parent of another node. This is only possible if n_2 is in passed, which means that n_2 was in wait and was not covered at some point. Since a zone can only be modified to be smaller, this means that n_2 has been modified at some point. Otherwise n_2 has always been covered by n_1, which is not possible. Since n_2 has been modified, and it is covered (at least by n_1), this means that cut has been called on n_2 last time its zone has been modified. This means that n_2 has no children and is not in passed, contradicting our assumption. Hence the algorithm always terminates. □

4 Symbolic Algorithm

4.1 Boolean encoding of zones

We now present a symbolic algorithm that represents abstract states using Boolean formulas. Let $\mathbb{B} = \{0, 1\}$, and $\mathcal{V}$ be a set of variables. A Boolean formula f that uses variables from set $X \subseteq \mathcal{V}$ will be written $f(X)$ to make the dependency explicit; we sometimes write $f(X, Y)$ in place of $f(X \cup Y)$. Such a formula represents a set $[\![f]\!] = \{v \in \mathbb{B}^{\mathcal{V}} \mid v \models f\}$. We consider primed versions of all variables; this will allow us to write formulas relating two valuations. For any subset $X \subseteq \mathcal{V}$, we define $X' = \{p' \mid p \in X\}$.

A *literal* is either p or $\neg p$ for a variable p. Given a set X of variables, an X-*minterm* is the conjunction of literals where each variable of X appears exactly once. X-minterms can be seen as elements of $\mathbb{B}^X$.

Given a vector of Boolean formulas $Y = (Y_x)_{x \in X}$, formula $f[Y/X]$ is the *substitution of X by Y in f*, obtained by replacing each $x \in X$ with the formula Y_x. The positive cofactor of $f(X)$ by x is $\exists x.\ (x \wedge f(X))$, and its negative cofactor is $\exists x.\ (\neg x \wedge f(X))$.

Let us define a generic operator $\texttt{post}$ that computes successors of a set $S(X, Y)$ given a relation $R(X, X')$ (here, Y designates any set of variables on which S might depend outside of X): $\texttt{post}_R(S(X, Y)) = (\exists X.S(X, Y) \wedge R(X, X'))[X/X']$. Similarly, we set $\texttt{pre}_R(S(X, Y)) = (\exists X'.S(X, Y)[X'/X] \wedge R(X, X'))$, which computes the predecessors of $S(X, Y)$ by the relation R [24].

Clock predicate abstraction. We fix a total order $\lhd$ on $\mathcal{C}_0$. In this section, abstract domains are defined as $\mathcal{D} = (\mathcal{D}_{x,y})_{x \lhd y \in \mathcal{C}_0}$, that is only for pairs $x \lhd y$. In fact, constraints of the form $x - y \leq k$ with $x \rhd y$ are encoded using the negation of $y - x < -k$ since $(x - y \leq k) \Leftrightarrow \neg(y - x < -k)$. We thus define $\mathcal{D}_{x,y} = -\mathcal{D}_{y,x}$ for all $x \rhd y$.

For $x, y \in \mathcal{C}_0$, let $\mathcal{P}_{x,y}$ denote the set of *clock predicates associated to $\mathcal{D}_{x,y}$*:

$$\mathcal{P}^{\mathcal{D}}_{x,y} = \{P_{x-y \prec k} \mid (k, \prec) \in \mathcal{D}_{x,y}\}.$$

Let $\mathcal{P}^{\mathcal{D}} = \cup_{x,y\in\mathcal{C}_0}\mathcal{P}_{x,y}$ denote the set of all clock predicates associated with $\mathcal{D}$ (we may omit the superscript $\mathcal{D}$ when it is clear). For all $(x,y) \in {\mathcal{C}_0}^2$ and $(k,\prec) \in \mathcal{D}_{x,y}$, we denote by $p_{x-y\prec k}$ the literal $P_{x-y\prec k}$ if $x \lhd y$, and $\neg P_{y-x\prec^{-1}-k}$ otherwise (where $\leq^{-1} = <$ and $<^{-1} = \leq$). We also consider a set $\mathcal{B}$ of Boolean variables used to encode locations. Overall, the state space is described using Boolean formulas on these two types of variables, so states are elements of $\mathbb{B}^{\mathcal{P}\cup\mathcal{B}}$.

Our Boolean encoding of clock constraints and semantic operations follow those of [27] for a concrete domain. We define these however for abstract domains, and show how all successor computation and refinement operations can be performed.

Let us define the *clock semantics* of predicate $P_{x-y\preceq k}$ as $[\![P_{x-y\preceq k}]\!]_{\mathcal{C}_0} = \{\nu \in \mathbb{R}_{\geq 0}^{\mathcal{C}_0} \mid \nu(x) - \nu(y) \preceq k\}$. Since the set $\mathcal{C}$ of clocks is fixed, we may omit the subscript and just write $[\![P_{x-y\preceq k}]\!]$. We define the conjunction, disjunction, and negation as intersection, union, and complement, respectively. Given a $\mathcal{P}$-minterm $v \in \mathbb{B}^{\mathcal{P}}$, we define $[\![v]\!]_{\mathcal{D}} = \bigcap_{p \text{ s.t. } v(p)} [\![p]\!]_{\mathcal{D}} \cap \bigcap_{p \text{ s.t. } \neg v(p)} [\![p]\!]_{\mathcal{D}}^c$. Thus, negation of a predicate encodes its complement. For a Boolean formula $F(\mathcal{P})$, we set $[\![F]\!] = \bigcup_{v\in\mathsf{Minterms}(F)} [\![v]\!]_{\mathcal{D}}$. Intuitively, the minterms of $\mathcal{P}$ define smallest zones of $\mathbb{R}_{\geq 0}^{\mathcal{C}}$ definable using $\mathcal{P}$. A minterm $v \in \mathbb{B}^{\mathcal{B}\cup\mathcal{P}}$ defines a pair $[\![v]\!]_{\mathcal{D}} = (l, Z)$ where l is encoded by $v_{|\mathcal{B}}$ and $Z = [\![v_{|\mathcal{P}}]\!]_{\mathcal{D}}$. A Boolean formula F on $\mathcal{B} \cup \mathcal{P}$ defines a set $[\![F]\!]_{\mathcal{D}} = \cup_{v\in\mathsf{Minterms}(F)} [\![v]\!]_{\mathcal{D}}$ of such pairs. A minterm v is *satisfiable* if $[\![v]\!]_{\mathcal{D}} \neq \emptyset$.

An abstract domain $\mathcal{D}$ induces an *abstraction function* $\alpha_{\mathcal{D}} \colon 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}} \to 2^{\mathbb{B}^{\mathcal{P}}}$ with $\alpha_{\mathcal{D}}(Z) = \{v \mid v \in \mathbb{B}^{\mathcal{P}} \text{ and } [\![v]\!]_{\mathcal{D}} \cap Z \neq \emptyset\}$, from the set of zones to the set of subsets of Boolean valuations on $\mathcal{P}$. We define the *concretization function* as $[\![\cdot]\!]_{\mathcal{D}} \colon 2^{\mathbb{B}^{\mathcal{P}}} \to 2^{\mathbb{R}_{\geq 0}^{\mathcal{C}}}$. The pair $(\alpha_{\mathcal{D}}, [\![\cdot]\!]_{\mathcal{D}})$ is a Galois connection, and $[\![\alpha_{\mathcal{D}}(Z)]\!]_{\mathcal{D}}$ is the most precise abstraction of Z in the domain induced by $\mathcal{D}$. Notice that $\alpha_{\mathcal{D}}$ is non-convex in general: for instance, if the clock predicates are $x \leq 2, y \leq 2$, then the set defined by the constraint $x = y$ maps to $(p_{x\leq 2} \wedge p_{y\leq 2}) \vee (\neg p_{x\leq 2} \wedge \neg p_{y\leq 2})$, which is not convex.

4.2 Reduction

We now define the reduction operation, which is similar to the reduction of DBMs. The idea is to eliminate unsatisfiable minterms from a given Boolean formula. For example, we would like to make sure that in all minterms, if $p_{x-y\leq 1}$ holds, then so does $p_{x-y\leq 2}$, when both are available predicates. Another issue is to eliminate minterms that are unsatisfiable due to triangle inequality. This is similar to the shortest path computation used to turn DBMs in canonical form.

Let a *path* in $\mathcal{D}$ be a sequence $x_1, (\alpha_1, \prec_1), x_2, (\alpha_2, \prec_2), \ldots, x_k, (\alpha_k, \prec_k), x_{k+1}$ where $x_1, \ldots, x_{k+1} \in \mathcal{C}_0$, and $(\alpha_i, \prec_i) \in \mathcal{D}_{x_i,x_{i+1}}$ for $1 \leq i \leq k$. Let us define $\mathsf{Paths}_k^{\mathcal{D}}(x - y \prec \alpha)$ as the set of paths from x to y of length k and weight at most $(\alpha, \prec)$, that is, paths $x_1, (\alpha_1, \prec_1), x_2, (\alpha_2, \prec_2), \ldots, x_k, (\alpha_k, \prec_k), x_{k+1}$ with $x_1 = x, x_{k+1} = y$, and $\sum_{i=1}^{k} (\alpha_i, \prec_i) \leq (\alpha, \prec)$. We also denote $\mathsf{Paths}_{\leq k}^{\mathcal{D}}(x - y \prec \alpha) = \bigcup_{l\leq k} \mathsf{Paths}_l^{\mathcal{D}}(x - y \prec \alpha)$. For a path $\pi = x_1, (\alpha_1, \prec_1), \ldots, x_{k+1}$ and minterm v, let us write $v \models \pi$ for the statement $v \models \wedge_{i=1}^{k} p_{x_i - x_{i+1} \prec \alpha_i}$.

A minterm $v \in \mathbb{B}^{\mathcal{P}}$ is *k-reduced* if for all $(x, y) \in {\mathcal{C}_0}^2$ and $(\alpha, \prec) \in \mathcal{D}_{x,y}$, for all $\pi = x_1, (\alpha_1, \prec_1), x_2, \ldots, x_{k+1} \in \mathsf{Paths}_{\leq k}(x - y \prec \alpha)$, whenever $v \models \pi$, we also have $v \models p_{x_1 - x_{k+1} \prec \alpha}$ for all $(\alpha, \prec) \in \mathcal{D}_{x_1, x_{k+1}}$ with $(\alpha, \prec) \geq \sum_{i=1}^{k} (\alpha_i, \prec_i)$. Thus, in a k-reduced minterm, no contradictions can be obtained via paths of length less than or equal to k. Observe that $|\mathcal{C}_0|$-reduction is equivalent to satisfiability since the condition then includes all paths, and it is known that in the absence of negative cycles, a set of difference constraints is satisfiable. Furthermore, for the concrete domain, k-reduction is equivalent to reduction for any $k \geq 2$. A formula is said to be k-reduced if all its minterms are k-reduced.

Example 1. Given predicates $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}, p_{x-z\leq 2}\}$, the formula $p_{x-y\leq 1} \wedge p_{y-z\leq 1}$ is not reduced since it contains the unsatisfiable minterm $p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge \neg p_{x-z\leq 2}$. However, the same formula is reduced if $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}\}$.

Consider now the predicate set $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}, p_{z-w\leq 3}, p_{x-w\leq 5}\}$, and consider the formula $\phi = p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge p_{z-w\leq 3}$ which is 2-reduced. Notice that the reduction of ϕ is $p_{x-y\leq 1} \wedge p_{y-z\leq 1} \wedge p_{z-w\leq 3} \wedge p_{x-w\leq 5}$ since the last predicate is implied by the conjunction of others. However, ϕ is 2-reduced since no path of length 2 allows to deduce $p_{x-w\leq 5}$. In the concrete domain, $p_{x-y\leq 1} \wedge p_{y-z\leq 1}$ would imply $p_{x-z\leq 2}$, thus, $p_{x-w\leq 5}$ could be derived too. It is indeed because of the abstract domain that 2-reduction might fail to capture all shortest paths.

In this paper, we only consider 2-reduction since computing reductions is the most expensive operation in our algorithms, and the formula below defining 2-reduction already tends to grow in size. Let us define $\mathsf{reduce}^2_{\mathcal{D}}$ as follows

$$\bigwedge_{\substack{(x,y) \in {\mathcal{C}_0}^2 \\ (k,\prec) \in \mathcal{D}_{x,y}}} \left[p_{x-y\prec k} \leftarrow \Big(\bigvee_{\substack{(l_1,\prec_1) \in \mathcal{D}_{x,y} \\ (l_1,\prec_1) \leq (k,\prec)}} p_{x-y\prec_1 l_1} \vee \bigvee_{\substack{z \in \mathcal{C}_0, (l_1,\prec_1) \in \mathcal{D}_{x,z}, \\ (l_2,\prec_2) \in \mathcal{D}_{z,y} \\ (l_1,\prec_1)+(l_2,\prec_2') \leq (k,\prec)}} p_{x-z\prec_1 l_1} \wedge p_{z-y\prec_2 l_2} \Big) \right]$$

The formula intuitively applies shortest paths over paths of length 1 or 2.

Lemma 10. *For all formulas $S(\mathcal{P})$, we have $[\![S]\!]_{\mathcal{D}} = [\![\mathsf{reduce}^2_{\mathcal{D}}(S)]\!]_{\mathcal{D}}$ and all minterms of $\mathsf{reduce}^2_{\mathcal{D}}(S)$ are 2-reduced.*

Proof. It is easy to see that $[\![\mathsf{reduce}^k_{\mathcal{D}}(S)]\!]_{\mathcal{D}} \subseteq [\![S]\!]_{\mathcal{D}}$. In fact, any minterm of the former is a minterm of S as well by definition.

To see the converse, consider $v \in \mathsf{Minterms}(S)$. We show that $[\![v]\!]_{\mathcal{D}} \subseteq [\![\mathsf{reduce}^k_{\mathcal{D}}(S)]\!]_{\mathcal{D}}$. If $[\![v]\!]_{\mathcal{D}} = \emptyset$ then the inclusion holds trivially. Otherwise, we must have $v \in \mathsf{Minterms}(\mathsf{reduce}^k_{\mathcal{D}}(S))$. In fact, we show that all minterms $v \notin \mathsf{Minterms}(\mathsf{reduce}^k_{\mathcal{D}}(S))$ satisfy $[\![v]\!] = \emptyset$. Consider such a minterm v. There must exist $x, y \in \mathcal{C}_0$ and $(k, \prec_k) \in \mathcal{D}_{x,y}$ such that $\neg p_{x-y\prec k}$ but the right hand side of the implication holds. But this implies that $[\![v]\!] = \emptyset$.

The fact that all miterms of $\mathsf{reduce}^k_{\mathcal{D}}(S)$ are k-reduced follow by the definition of the operator. $\square$

Example 2. Note that $\mathsf{reduce}^2_{\mathcal{D}}(S)$ can still contain valuations that are unsatisfiable. Consider $\mathcal{P} = \{p_{x-y\leq 1}, p_{y-z\leq 1}, p_{x-z\leq 4}, p_{z-x\leq -3}\}$. Then the minterm u

that sets all predicates to true is still contained in $\mathsf{reduce}^2_{\mathcal{D}}(S)$ although $[\![u]\!] = \emptyset$ since $x - y \leq 1 \wedge y - z \leq 1$ implies $x - z \leq 2$ which contradicts $z - x \leq -3$. Here, adding the predicate $p_{x-z\leq 2}$ or $p_{x-z<3}$ would render the abstraction precise enough to eliminate this valuation in $\mathsf{reduce}^2_{\mathcal{D}}(S)$.

Let us see how an abstraction can be refined so that the reduced constraint eliminates a given unsatisfiable minterm.

Lemma 11. *Let $v \in \mathbb{B}^{\mathcal{P}^{\mathcal{D}}}$ be a minterm such that $v \models \mathsf{reduce}^2_{\mathcal{D}}$ and $[\![v]\!] = \emptyset$. One can compute in polynomial time a refinement $\mathcal{D}' \supset \mathcal{D}$ such that $v \not\models \mathsf{reduce}^2_{\mathcal{D}'}$.*

Proof. Consider a (non-canonial) DBM D that encodes v. Formally, for all $(x, y) \in {\mathcal{C}_0}^2$, $D(x, y) = \min\{(k, \prec) \mid p_{x-y\prec k} \in \mathcal{P}_{x,y}$ and $v(p_{x-y\prec k}) = 1\}$. The corresponding graph must have a negative cycle $s_1 s_2 \ldots s_m$ with $s_m = s_1$. To define $\mathcal{D}'$, we add the following predicates to $\mathcal{D}$:

- $s_i - s_{i+1} \leq D(s_i, s_{i+1})$ for all $1 \leq j \leq m - 1$.
- $s_1 - s_j \leq D(s_1, s_2) + D(s_2, s_3) + \ldots + D(s_{j-1}, s_j)$ for $1 \leq j \leq m - 1$.

Intuitively, along the negative cycle, we are adding predicates to represent exactly each single step, and also each big step from s_1 to s_j. This allows to derive the negative cycle using only paths of length 2.

More precisely, $v \wedge \mathsf{reduce}^2_{\mathcal{D}'}$ implies the following two predicates: $s_1 - s_m \leq \sum_{j=1}^{m-1} D(s_j, s_{j+1})$, and $s_m - s_1 \leq D(s_m, s_1)$ as implied by v. Since $-D(s_m, s_1) > \sum_{j=1}^{m-1} D(s_j, s_{j+1})$ (due to the negative cycle), the first predicate entails that $\neg p_{s_1 - s_m \leq -D(s_m, s_1)}$, which contradicts the second one. Hence $v \not\models \mathsf{reduce}^2_{\mathcal{D}'}$. □

4.3 Successor Computation

In this section, we explain how successor computation is realized in our encoding. For a guard g, assume we have computed an abstraction $\alpha_{\mathcal{D}}(g)$ in the present abstract domain. For each transition $\sigma = (\ell_1, g, R, \ell_2)$, let us define the formula $T_\sigma = \ell_1 \wedge \alpha_{\mathcal{D}}(g)$. We show how each basic operation on zones can be computed in our BDD encoding. In our algorithm, all formulas $A(\mathcal{B}, \mathcal{P})$ representing sets of states are assumed to be reduced, that is, $A(\mathcal{B}, \mathcal{P}) \subseteq \mathsf{reduce}^2_{\mathcal{D}}(A(\mathcal{B}, \mathcal{P}))$.

The intersection operation is simply logical conjunction.

Lemma 12. *For all reduced formulas $A(\mathcal{P}), B(\mathcal{P})$, $A(\mathcal{P}) \wedge B(\mathcal{P}) = \alpha_{\mathcal{D}}([\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}})$.*

Proof. Consider $v \in \mathsf{Minterms}(A(\mathcal{P}) \wedge B(\mathcal{P}))$. Then $v \in \mathsf{Minterms}(A(\mathcal{P})) \cap \mathsf{Minterms}(B(\mathcal{P}))$. So $[\![v]\!] \subseteq [\![A(\mathcal{P})]\!] \cap [\![B(\mathcal{P})]\!]$, and $v = \alpha_{\mathcal{D}}([\![v]\!]) \subseteq \alpha_{\mathcal{D}}([\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}})$.

We will now show $\alpha_{\mathcal{D}}([\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}}) \subseteq A(\mathcal{P}) \wedge B(\mathcal{P})$, which is equivalent to $[\![A(\mathcal{P})]\!]_{\mathcal{D}} \cap [\![B(\mathcal{P})]\!]_{\mathcal{D}} \subseteq [\![A(\mathcal{P}) \wedge B(\mathcal{P})]\!]$. Consider any clock valuation ν in the LHS, and let $v = \alpha_{\mathcal{D}}(\nu)$. Since $\nu \in [\![A(\mathcal{P})]\!]_{\mathcal{D}}$ and $\nu \in [\![B(\mathcal{P})]\!]_{\mathcal{D}}$, we must have $v \in A(\mathcal{P}) \wedge B(\mathcal{P})$, so $\nu \in [\![A(\mathcal{P}) \wedge B(\mathcal{P})]\!]$. □

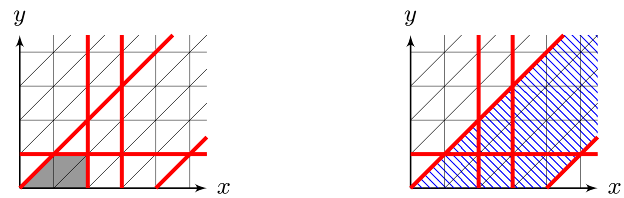


Fig. 4: Time successors: the successors of the gray zone on the left are the union of all blue dashed zones on the right.

For the time successors, we define

$$S_{\mathsf{Up}} = \bigwedge_{\substack{x \in \mathcal{C} \\ (k, \prec) \in \mathcal{D}_{x,0}}} (\neg p_{x-0\prec k} \rightarrow \neg p'_{x-0\prec k}) \bigwedge_{\substack{x,y \in \mathcal{C}_0, x \neq 0 \\ (k,\prec) \in \mathcal{D}_{x,y}}} (p'_{x-y\prec k} \leftrightarrow p_{x-y\prec k}).$$

Note that this relation is not a function: for $x \leq 0, y = 0$, if $\neg p_{x-y\prec k}$, then necessarily $\neg p'_{x-y\prec k}$; but otherwise both truth values for $p'_{x-y\prec k}$ are allowed. In fact, the formula only says that all lower bounds on clocks and diagonal constraints must be preserved. We let $\mathsf{Up}(A(\mathcal{B}, \mathcal{P})) = \mathsf{reduce}(\mathtt{post}_{S_{\mathsf{Up}}}(A(\mathcal{B}, \mathcal{P})))$.

Example 3. Figure 4 shows this operation applied to the gray zone on the left. The overapproximation is visible in this example. In fact, the blue dashed zone defined by $Z = 3 < x < 5 \land 0 \leq y < 1 \land x - y < 4$ (on the bottom right) is computed as a successor of the gray zone although no point of the gray zone can actually reach Z by a time delay. Adding more diagonal constraints to the abstract domain, for instance, $x - y \leq 2$ would eliminate this successor.

Lemma 13. *For any Boolean formula $A(\mathcal{B}, \mathcal{P})$, $\alpha_{\mathcal{D}}(\llbracket A \rrbracket\uparrow) \subseteq \mathsf{Up}(A)$. Moreover, if $\mathcal{D}$ is the concrete domain and A is reduced, then this holds with equality.*

Proof. We show that $\alpha_{\mathcal{D}}(\llbracket A \rrbracket\uparrow) \subseteq \mathsf{Up}(A)$, which is equivalent to $\llbracket A \rrbracket\uparrow \subseteq \llbracket \mathsf{Up}(A) \rrbracket_{\mathcal{D}}$. Let $\nu' \in \llbracket A \rrbracket\uparrow$, and let $\nu \in \llbracket A \rrbracket$ such that $\nu' = \nu + d$ for some $d \geq 0$. We write $u = \alpha_{\mathcal{D}}(\nu) \in \mathsf{Minterms}(A)$, and $v = \alpha_{\mathcal{D}}(\nu')$. We now show that $(u, v) \in S_{\mathsf{Up}}$. This suffices to prove the inclusion since $v \in \mathsf{Minterms}(\mathsf{Up}(A))$ implies that $\nu' \in \llbracket v \rrbracket_{\mathcal{D}} \subseteq \llbracket \mathsf{Up}(A) \rrbracket_{\mathcal{D}}$. Observe that $\alpha_{\mathcal{D}}(\nu)$ and $\alpha_{\mathcal{D}}(\nu')$ satisfy the same diagonal predicates of the form $p_{x-y\prec\alpha}$ with $x, y \neq 0$, since $\nu' = \nu + d$. Moreover, any lower bound satisfied by u is also satified by v. Thus, $(u, v) \in S_{\mathsf{Up}}$.

Assume now that A is reduced. We show that $\mathsf{Up}(A) \subseteq \alpha_{\mathcal{D}}(\llbracket A \rrbracket\uparrow)$. Let $v \in \mathsf{Minterms}(\mathsf{Up}(A))$, and let $(u, v) \in S_{\mathsf{Up}}$ with $u \in \mathsf{Minterms}(A)$. We claim that $\llbracket v \rrbracket \subseteq \llbracket u \rrbracket\uparrow$. This suffices to prove the inclusion since $\{v\} = \alpha_{\mathcal{D}}(\llbracket v \rrbracket) \subseteq \alpha_{\mathcal{D}}(\llbracket u \rrbracket\uparrow) \subseteq \alpha_{\mathcal{D}}(\llbracket A \rrbracket\uparrow)$. To prove the claim, it suffices to see $\llbracket u \rrbracket$ and $\llbracket v \rrbracket$ as DBMs. In fact, S_{Up} precisely corresponds to the up operation on DBMs. In particular, u and v have the same diagonal constraints, and any lower bound of u is a lower bound of v. Because u is reduced, this implies that $\llbracket v \rrbracket\uparrow \subseteq \llbracket u \rrbracket$. $\square$

Example 4. Note that we do need that the domain is concrete to prove equality. In fact, assume the predicates are $p_1 = y-x \le 3, p_2 = x-y \le 1, x \le 1, y \le 2, x \le 2, y \le 4$. Then if $u = x \le 1 \wedge y \le 2 \wedge p_1 \wedge p_2$ and $v = 1 \le x \le 2 \wedge y \le 4 \wedge p_1 \wedge p_2$. So v is larger than $u\uparrow$.

Second, to see that the Up operation can yield strict over-approximations, consider $p_{x\le 1} \wedge p_{y\ge 1}$ in a domain with no predicate on $x - y$. The operator relaxes all upper bounds, yielding $p_{y\ge 1}$ which defines a set larger than the concrete time-successors $y \ge 1 \wedge x - y \le 0$.

Alternatively, we can also use the method described in [27, Theorem 2] to compute time successors, but the above relation will allow us to compute predecessors as well.

Last, we define the reset operation as follows. For any $z \in \mathcal{C}$,

$$\begin{aligned}
S_{\mathsf{Reset}_z} = & \bigwedge_{(0,\le)\le(k,\prec)\in\mathcal{D}_{z,0}} p'_{z-0\prec k} \\
\wedge & \bigwedge_{\substack{x\neq z\\(k,\prec)\in\mathcal{D}_{x,z}}} \Big(\bigvee_{\substack{(l,\prec')\in\mathcal{D}_{x,0}\\(l,\prec')\le(k,\prec)}} p_{x-0\prec' l}\Big) \Rightarrow p'_{x-z\prec k} \\
\wedge & \bigwedge_{\substack{y\neq z\\(k,\prec)\in\mathcal{D}_{z,y}}} \Big(\bigvee_{\substack{(l,\prec')\in\mathcal{D}_{0,y}\\(l,\prec')\le(k,\prec)}} p_{0-y\prec' l}\Big) \Rightarrow p'_{z-y\prec k} \\
\wedge & \bigwedge_{\substack{y\neq z\\(0,\le)\le(k,\prec)\in\mathcal{D}_{z,y}}} p'_{z-y\prec k} \\
\wedge & \bigwedge_{\substack{x,y\neq z\\(k,\prec)\in\mathcal{D}_{x,y}}} p'_{x-y\prec k} \Leftrightarrow p_{x-y\prec k}.
\end{aligned}$$

Intuitively, the first conjunct ensures all non-negative upper bounds on the reset clock hold; the second conjunct ensures that a diagonal predicate $x - z \prec k$ with $x \neq z$ is set to true if, and only if, an upper bound $(l, \prec') \le (k, \prec)$ already holds on x. Recall that in operations on DBMs, one sets such a diagonal component to the tightest upper bound on x. The third conjunct is symmetric to the second, and the last one ensures diagonals not affected by reset are unchanged. Let us define $\mathsf{Reset}_z(A) = \mathsf{reduce}(\mathtt{post}_{S_{\mathsf{Reset}_z}}(A))$.

Lemma 14. *For any Boolean formula $A(\mathcal{B}, \mathcal{P})$, and for any $z \in \mathcal{C}$, we have $\alpha_{\mathcal{D}}(\mathtt{Reset}_z([\![A]\!]_{\mathcal{D}})) \subseteq \mathsf{Reset}_z(A)$. Moreover, if $\mathcal{D}$ is the concrete domain, and A is reduced, then the above holds with equality.*

Proof. Let $r = \{z\}$. We show the equivalent inclusion $\mathtt{Reset}_r([\![A]\!]_{\mathcal{D}}) \subseteq [\![\mathsf{Reset}_r(A)]\!]_{\mathcal{D}}$. Let $\nu' \in \mathtt{Reset}_r([\![A]\!])$, and $\nu \in [\![A]\!]$ such that $\nu' = \nu[r \leftarrow 0]$. So $u = \alpha_{\mathcal{D}}(\nu) \in A$. Letting $v = \alpha_{\mathcal{D}}(\nu')$, let us show that $(u, v) \in S_{\mathsf{Reset}_r}$, which proves that $v \in \mathsf{Reset}_r(A)$, thus $\nu' \in [\![v]\!]_{\mathcal{D}} \subseteq [\![\mathsf{Reset}_r(A)]\!]_{\mathcal{D}}$.

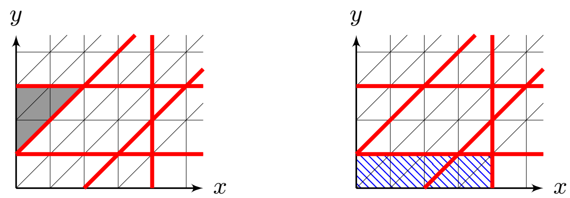


Fig. 5: Reset Operation: The abstract successor of the gray zone on the left is the blue dashed zone on the right.

- Consider $x \in r$. Since $\nu'(x) = 0$, we have that $v \models p_{x-y\prec k}$ for all $(k, \prec) \in \mathcal{D}_{x,y}$ with $(k, \prec) \geq (0, \leq)$; so (u, v) satisfies the first conjunct. Fix $(k, \prec) \in \mathcal{D}_{x,y}$.
- Assume $x \notin r, y \in r$, so that $\nu'(x) = \nu(x), \nu'(y) = 0$. If there is $(l, \prec') \in \mathcal{D}_{x,0}$ with $(l, \prec') \leq (k, \prec)$, this means $\nu(x) - 0 \prec' l \prec k$ so $\nu'(x) - \nu'(y) \prec k$. Thus the second conjunct is satisfied.
- Assume $x \in r, y \notin r$ so that $\nu'(x) = 0, \nu'(y) = \nu(y)$. If there is $(l, \prec') \in \mathcal{D}_{0,y}$ with $(l, \prec') \leq (k, \prec)$, this means $0 - \nu(y) \prec' l \prec k$ so $\nu'(x) - \nu'(y) \prec k$ as well. Thus the third conjunct is satisfied. We also have trivially $\nu'(x) - \nu'(y) \prec k$ for all $(0, \leq) \leq (k, \prec)$, which entails the fourth conjunct.
- Observe that ν and ν' satisfy the same predicates of type $p_{x-y\prec\alpha}$ with $x, y \notin r$ since these values are not affected by the reset; so the pair (u, v) satisfies the last conjunct of S_{Reset_r} as well.

Now, if the domain is concrete and A is reduced, then we have $[\![\mathsf{Reset}_r(A)]\!]_{\mathcal{D}} \subseteq \mathtt{Reset}_r([\![A]\!]_{\mathcal{D}})$. In fact, the operation then corresponds precisely to the reset operation in DBMs, see [8, Algorithm 10]. On DBMs, for a component (x, y) with $x \in r, y \notin r$, the algorithm consists in setting this component to value $(0, y)$. In our encoding, we thus set the predicate $p'_{x-y\prec k}$ to true whenever $p_{0-x\prec' l}$ holds with $(l, \prec') \leq (k, \prec)$. The argument is symmetric for (x, y) with $x \notin r$, $y \in r$. $\square$

4.4 Model-checking algorithm

Algorithm 8 shows how to check the reachability of a target location given an abstract domain. The list layers contains, at position i, the set of states that are reachable in i steps. The function $\mathsf{ApplyEdges}$ computes the disjunction of immediate successors by all edges. It consists in looping over all edges $e = (l_1, g, R, l_2)$, and gathering the following image by e:

$$\mathsf{enc}(\ell_2) \wedge \mathsf{Reset}_{r_k}(\mathsf{Reset}_{r_{k-1}}(\ldots(\mathsf{Reset}_{r_1}((((\exists \mathcal{B}.A(\mathcal{B}, \mathcal{P}) \wedge \mathsf{enc}(\ell_1)) \wedge \alpha_{\mathcal{D}}(g))))))),$$

where $R = \{r_1, \ldots, r_k\}$. We thus use a partitioned transition relation and do not compute the monolithic transition relation.

When the target location is found to be reachable, $\mathsf{ExtractTrace}(\mathsf{layers})$ returns a trace reaching the target location. This is standard and can be done by computing backwards from the last element of layers, by finding which edge

Algorithm 8: Algorithm SymReach that checks the reachability of a target location l_T in a given abstract domain $\mathcal{D}$.

Input: $\mathcal{A} = (\mathcal{L}, \mathsf{Inv}, \ell_0, \mathcal{C}, E)$, ℓ_T, $\mathcal{D}$

;
next := $\mathsf{enc}(l_0) \wedge \alpha_{\mathcal{D}}(\wedge_{x \in \mathcal{C}} x = 0)$;
layers := [];
reachable := `false`;
while ($\neg$reachable $\wedge$ next) $\neq$ `false` **do**
    reachable := reachable $\vee$ next;
    next := ApplyEdges(Up(next)) $\wedge$ $\neg$reachable;
    layers.$push$(next);
    if (next $\wedge$ enc(l_T)) $\neq$ `false` **then**
        return ExtractTrace (layers);
return Not reachable;

can be applied to reach the current state. Since both reset and time successor operations are defined using relations, predecessors in our abstract system can be easily computed using the operator $\mathtt{pre}_R$. As it is standard, we omit the precise definition of this function (the reader can refer to the implementation) but assume that it returns a trace of the form

$$A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{n-1}} A_n,$$

where the $A_i(\mathcal{B}, \mathcal{P})$ are minterms and the σ_i belong to the trace alphabet $\Sigma = \{\mathrm{up}, r_\emptyset\} \cup \{r(x)\}_{x \in \mathcal{C}}$, with the following meaning:

- if $A_i \xrightarrow{\mathrm{up}} A_{i+1}$ then $A_{i+1} = \mathsf{Up}(A_i)$;
- if $A_i \xrightarrow{r_\emptyset} A_{i+1}$ then $A_{i+1} = A_i$;
- if $A_i \xrightarrow{r(x)} A_{i+1}$ then $A_{i+1} = \mathsf{Reset}_x(A_i)$.

The feasibility of such a trace is easily checked using DBMs.

The overall algorithm then follows a classical CEGAR scheme. We initialize $\mathcal{D}$ by adding the clock constraints that appear syntactically in $\mathcal{A}$, which is often a good heuristic. We run the reachability check of Algorithm 8. If no trace is found, then the target location is not reachable. If a trace is found, then we check for feasibility. If it is feasible, then the counterexample is confirmed. Otherwise, the trace is spurious and we run the refinement procedure described in the next subsection, and repeat the analysis.

4.5 Abstraction refinement

Since we initialize $\mathcal{D}$ with all clock constraints appearing in guards, we can make the following hypothesis.

Assumption 1. *All guards are represented exactly in the considered abstractions.*

Note that the algorithm can be easily extended to the general case; but this simplifies the presentation.

The abstract transition relation we use is not the most precise abstraction of the concrete transition relation. Therefore, it is possible to have abstract transitions $A_1 \xrightarrow{a} A_2$ for some action a while no concrete transition exists between $[\![A_1]\!]$ and $[\![A_2]\!]$. This requires care and is not a direct application of the standard refinement technique from [11]. A second difficulty is due to incomplete reduction of the predicates using $\mathsf{reduce}^2_{\mathcal{D}}$. In fact, some reachable states in our abstract model will be unsatisfiable. Let us explain how we refine the abstraction in each of these cases.

Consider an algorithm interp that returns an interpolant of two given zones Z_1, Z_2. In what follows, by the *refinement of* $\mathcal{D}$ *by* $\mathsf{interp}(Z_1, Z_2)$, we mean the domain $\mathcal{D}'$ obtained by adding $(k, \prec)$ to $\mathcal{D}_{x,y}$ for all constraints $x - y \prec k$ of $\mathsf{interp}(Z_1, Z_2)$. Observe that $\alpha_{\mathcal{D}'}(Z_1) \cap \alpha_{\mathcal{D}'}(Z_2) = \emptyset$ in this case.

We define concrete successor and predecessor operations for the actions in Σ. For each $a \in \Sigma$, let Pre^c_a denote the concrete predecessor operation on zones defined straightforwardly, and similarly for Post^c_a.

Consider domain $\mathcal{D}$ and the induced abstraction function $\alpha_{\mathcal{D}}$. Assume that we are given a spurious trace $\pi = A_1 \xrightarrow{\sigma_1} A_2 \xrightarrow{\sigma_1} \ldots \xrightarrow{\sigma_{n-1}} A_n$. Let $B_1 \ldots B_n$ be the sequence of concrete states visited along π in $\mathcal{A}$, that is, B_1 is the concrete initial state, and for all $2 \leq i \leq n$, let $B_i = \mathsf{Post}^c_{\pi_{i-1}}(B_{i-1})$. This sequence can be computed using DBMs.

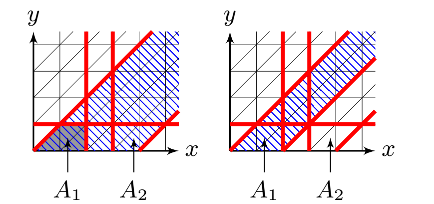
The trace is *realizable* if $B_n \neq \emptyset$, in which case the counterexample is confirmed. Otherwise it is *spurious*. We show how to refine the abstraction to eliminate a spurious trace π.

Let i_0 be the maximal index such that $B_{i_0} \neq \emptyset$. There are three possible reasons explaining why B_{i_0+1} is empty:

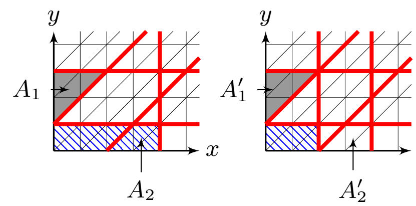
1. first, if the abstract successor A_{i_0+1} is unsatisfiable, that is, if it contains contradictory predicates; in this case, $[\![A_{i_0+1}]\!] = \emptyset$, and the abstraction is refined by Lemma 11 to eliminate this case by strengthening $\mathsf{reduce}^k_{\mathcal{D}}$.
2. if there are predecessors of A_{i_0+1} inside A_{i_0} but none of them are in B_{i_0}, i.e., $\mathsf{Pre}^c_{\pi_{i_0}}([\![A_{i_0+1}]\!]) \cap [\![A_{i_0}]\!] \neq \emptyset$; in this case, we refine the domain by separating these predecessors from the rest of A_{i_0} using $\mathsf{interp}(\mathsf{Pre}^c_{\pi_{i_0}}([\![A_{i_0+1}]\!]), B_{i_0-1})$, as in [11].
3. otherwise, there are no predecessors of A_{i_0+1} inside A_{i_0}: we refine the abstraction according to the type of the transition from step i_0 to i_0+1:
   (a) if $\pi_{i_0} = \text{up}$: refine $\mathcal{D}$ by $\mathsf{interp}([\![A_{i_0}]\!]{\uparrow}, [\![A_{i_0+1}]\!]{\downarrow})$.
   (b) if $\pi_{i_0} = r(x)$: refine $\mathcal{D}$ by $\mathsf{interp}(\mathsf{Free}_x([\![A_{i_0}]\!]), \mathsf{Free}_x([\![A_{i_0+1}]\!]))$.

Note that the case $\pi_{i_0} = r_\emptyset$ is not possible since this induces the identity function both in the abstract and concrete systems.

Given abstraction $\alpha_{\mathcal{D}}$ and spurious trace π, let $\mathsf{refine}(\alpha_{\mathcal{D}}, \pi)$ denote the refined abstraction $\alpha_{\mathcal{D}'}$ obtained as described above. The following two lemmas justify the two subcases of the third case above. They prove that the detected spurious transition disappears after refinement. The reset and up operations depend on



(a) Refinement for the time successors operation. The interpolant that separates $[\![A_1]\!]\uparrow$ from $[\![A_2]\!]\downarrow$ contains the constraint $x = y + 2$. When this is added to the abstract domain, the set A_2' (which is A_2 in the new abstraction) is no longer reachable by the time successors operation.



(b) Refinement for the reset operation. The interpolant that separates $\mathsf{Free}_y(A_1)$ from $\mathsf{Free}_y(A_2)$ contains the constraint $x < 2$. When this is added to the abstract domain, the set A_2' (which is A_2 in the new abstraction) is no longer reachable by the reset operation.

the abstraction, so we make this dependence explicit below by using superscripts, as in Reset_x^α and Up^α, in order to distinguish the operations before and after a refinement.

Lemma 15. *Consider* $(A_1, A_2) \in \mathsf{Up}^\alpha$ *with* $[\![A_1]\!]\uparrow \cap [\![A_2]\!] = \emptyset$. *Then* $[\![A_1]\!]\uparrow \cap [\![A_2]\!]\downarrow = \emptyset$. *Moreover, if* α' *is obtained by refinement of* α *by* $\mathsf{interp}([\![A_1]\!]\uparrow, [\![A_2]\!]\downarrow)$, *then for all* $(A_1', A_2') \in \mathsf{Up}^{\alpha'}$, $[\![A_1']\!] \subseteq [\![A_1]\!]$ *implies* $[\![A_2']\!] \cap [\![A_2]\!] = \emptyset$.

Proof. Assume there is $v \in [\![A_1]\!]\uparrow \cap [\![A_2]\!]\downarrow$. There exists $d_1, d_2 \geq 0$ and $v_1 \in A_1$ such that $v = v_1 + d_1$ and $v + d_2 \in A_2$, which means $v_1 + d_1 + d_2 \in A_2$, thus $[\![A_1]\!]\uparrow \cap [\![A_2]\!] \neq \emptyset$.

Let $Z = [\![\mathsf{interp}([\![A_1]\!]\uparrow, [\![A_2]\!]\downarrow)]\!]$. By definition, $[\![A_1]\!]\uparrow \subseteq Z$, and $Z \cap [\![A_2]\!]\downarrow = \emptyset$. We have $Z\uparrow = Z$; in fact, Z cannot have upper bounds since $[\![A_1]\!]\uparrow$ does not have any. It follows that $Z\uparrow \cap [\![A_2]\!]\downarrow = \emptyset$.

Now, consider $(A_1', A_2') \in \mathsf{Up}^{\alpha'}$ with $[\![A_1']\!] \subseteq [\![A_1]\!]$. Let us show that $[\![A_2']\!] \subseteq Z$. Notice that all constraints of Z must be satisfied by A_2' due to the inclusion $[\![A_1']\!] \subseteq Z$: there are no upper bounds in Z, all its lower bounds are satisfied by A_1, thus also by A_1', and are preserved by definition by Up, and all diagonal constraints of Z hold in A_1, thus also in A_1', and are preserved as well in A_2'. This shows the required inclusion. It follows that $[\![A_2']\!] \cap [\![A_2]\!] = \emptyset$. ◻

Lemma 16. *Consider* $x \in \mathcal{C}$, *and* $(A_1, A_2) \in \mathsf{Reset}_x^\alpha$ *such that* $[\![A_1]\!][x \leftarrow 0] \cap [\![A_2]\!] = \emptyset$. *Then* $\mathsf{Free}_x([\![A_1]\!]) \cap \mathsf{Free}_x([\![A_2]\!]) = \emptyset$. *Moreover, if* α' *is obtained by refinement of* α *by* $\mathsf{interp}(\mathsf{Free}_x([\![A_1]\!]), \mathsf{Free}_x([\![A_2]\!]))$, *then for all* $(A_1', A_2') \in \mathsf{Reset}_x^{\alpha'}$ *with* $[\![A_1']\!] \subseteq [\![A_1]\!]$, *we have* $[\![A_2']\!] \cap [\![A_2]\!] = \emptyset$.

Proof. Let $v \in \mathsf{Free}_x([\![A_1]\!]) \cap \mathsf{Free}_x([\![A_2]\!])$. Then there exist $v_1 \in [\![A_1]\!]$, $v_2 \in [\![A_2]\!]$ and v_0 such that $v_0 = v[x \leftarrow 0] = v_1[x \leftarrow 0] = v_2[x \leftarrow 0]$. But $[\![A_2]\!]$ is closed by resetting x, that is, $[\![A_2]\!][x := 0] \subseteq [\![A_2]\!]$. This follows from Lemma 14 applied to A_2, by observing that A_2 is unchanged by the reset operation. So $v_0 \in [\![A_2]\!]$. But then $[\![A_1]\!][x := 0] \cap [\![A_2]\!] \neq \emptyset$ as witnessed by $v_1[x := 0] = v_0$.

Let $Z = [\![\mathsf{interp}(\mathsf{Free}_x([\![A_1]\!]), \mathsf{Free}_x([\![A_2]\!]))]\!]$. For all A'_1 satisfying $[\![A'_1]\!] \subseteq [\![A_1]\!]$, we have $[\![A'_1]\!][x \leftarrow 0] \subseteq [\![A_1]\!][x \leftarrow 0] \subseteq \mathsf{Free}_x([\![A_1]\!]) \subseteq Z$. So $Z \cap A_2 = \emptyset$ means that $[\![A'_1]\!][x \leftarrow 0] \cap A_2 = \emptyset$. □

5 Experiments

We implemented both algorithms. The symbolic version was implemented in OCaml using the `CUDD` library[2]; the explicit version was implemented in C++ within an existing model checker using Uppaal DBM library. Both prototypes take as input networks of timed automata with invariants, discrete variables, urgent and committed locations. The presented algorithms are adapted to these features without difficulty.

We evaluated our algorithms on three classes of benchmarks we believe are significant. We compare the performance of the algorithm with that of Uppaal [7] which is based on zones, as well as the BDD-based model checker engine of PAT [25]. We were unable to compare with RED [29] which is not maintained anymore and not open source, and with which we failed to obtain correct results. The tool used in [16] was not available either. We thus only provide a comparison here with two well-maintained tools.

Two of our benchmarks are variants of schedulability-analysis problems where task execution times depend on the internal states of executed processes, so that an analysis of the state space is necessary to obtain a precise answer.

Monoprocess Scheduling Analysis. In this variant, a single process sequentially executes tasks on a single machine, and the execution time of each cycle depends on the state of the process. The goal is to determine a bound on the maximum execution time of a single cycle. This depends on the semantics of the process since the bound depends on the reachable states.

More precisely, we built a set of benchmarks where the processes are defined by synchronous circuit models taken from the Synthesis Competition (`http://www.syntcomp.org`). We assume that each latch of the circuit is associated with a resource, and changing the state of the resource takes some amount of time. So a subset of the latches have clocks associated with them, which measure the time elapsed since the latest value change (latest moment when the value changed from 0 to 1, or from 1 to 0). We provide two time positive bounds ℓ_0 and ℓ_1 for each latch, which determine the execution time as follows: if the value of latch ℓ changes from 0 to 1 (resp. from 1 to 0), then the execution time of the present cycle cannot be less than ℓ_1 (resp. ℓ_0). The execution time of the step is then the minimum that satisfies these constraints.

Multi-process Stateful Scheduling Analysis. In this variant, three processes are scheduled on two machines with a round-robin policy. Processes schedule tasks one after the other without any delay. As in the previous benchmarks, a process

[2] `http://vlsi.colorado.edu/~fabio/`

executing a task (on any machine) corresponds to a step of the synchronous circuit model. Each task is described by a tuple (C_1, C_2, D) which defines the minimum and maximum execution times, and the relative deadline. When a task finishes, the next task arrives immediately. The values in the tuple depend on the state of the process. The goal is to check the absence of any deadline miss. Processes are also instantiated with AIG circuits from `http://www.syntcomp.org`.

Asynchronous Computation. We consider an asynchronous network of "threshold gates", defined as follows: each gate is characterized by a tuple $(n, \theta, [l, u])$ where n is the number of inputs, $0 \leq \theta \leq n$ is the threshold, and $l \leq u$ are lower and upper bounds on activation time. Each gate has an output which is initially undefined. The gate becomes active during the time period $[l, u]$. During this time, if all inputs are defined, and if at least θ of the inputs have value 1, then it sets its output to 1. At the end of the time period, it becomes deactivated and the output becomes undefined again, until the next period, which starts l time units after the deactivation. The goal is to check whether the given gate can output 1 within a given time bound T.

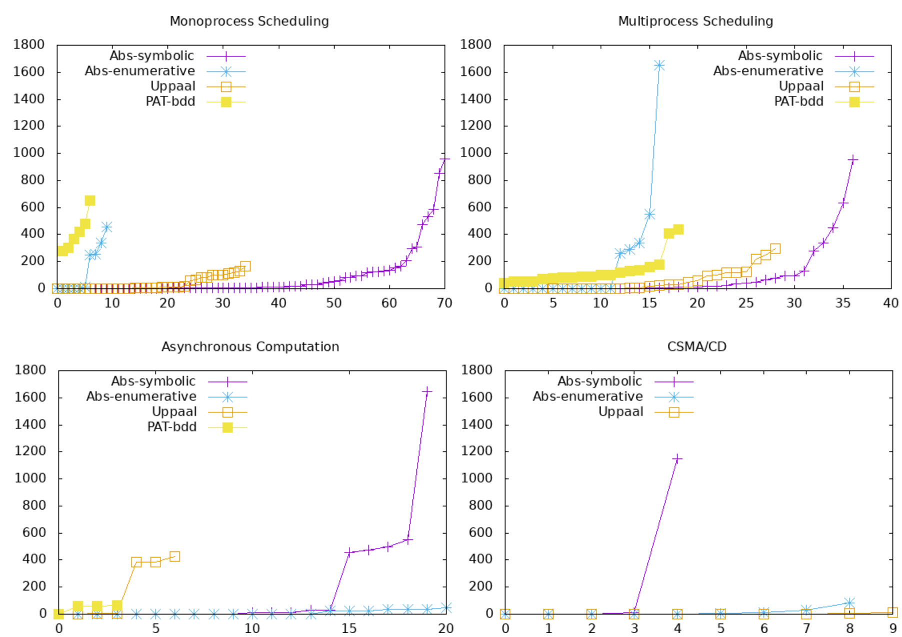


Fig. 7: Comparison of our enumerative and symbolic algorithms (resp. Abs-enumerative and Abs-symbolic) with Uppaal and PAT. Each figure is a cactus plot for the set of benchmarks: a point (X,Y) means X benchmarks were solved within time bound Y.

Results. Figure 7 displays the results of our experiments. All algorithms were given 8GB of memory and a timeout of 30 minutes, and the experiments were run on laptop with an Intel i7@3.2Ghz processor running Linux. The symbolic algorithm performs best among all on the monoprocess and multiprocess scheduling benchmarks. Uppaal is the second best, but does not solve as many benchmarks as our algorithm. Our enumerative algorithm quickly fails on these benchmarks, often running out of memory. On asynchronous computation benchmarks, our enumerative algorithm performs remarkably well, beating all other algorithms. We ran our tools on the CSMA/CD benchmarks (with 3 to 12 processes); Uppaal performs the best but our enumerative algorithm is slightly behind. The symbolic algorithm does not scale, while PAT fails to terminate in all cases.

The tool used for the symbolic algorithm is open source and can be found at `https://github.com/osankur/symrob` along with all the benchmarks.

6 Conclusion and Future Work

There are several ways to improve the algorithm. Since the choice of interpolants determines the abstraction function and the number of refinements, we assumed that taking the minimal interpolant should be preferable as it should keep the abstractions as coarse as possible. But it might be better to predict which interpolant is the most adapted for the rest of the computation in order to limit future refinements. The number of refinement also depends on the search order, and although it has already been studied in [23], it could be interesting to study it in this case. Generally speaking, it is worth noting that we currently cannot predict which (variant of) our algorithms is better suited for which model.

Several extensions of our algorithms could be developed, *e.g.* combining our algorithms with other methods based on finer abstractions as in [22], integrating predicate abstraction on discrete variables, or developing SAT-based versions of our algorithms.

References

1. E. Althaus, B. Beber, W. Damm, S. Disch, W. Hagemann, A. Rakow, C. Scholl, U. Waldmann, and B. Wirtz. Verification of linear hybrid systems with large discrete state spaces using counterexample-guided abstraction refinement. *Science of Computer Programming*, 148:123–160, Nov. 2017.
2. R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.
3. R. Alur and D. L. Dill. Automata for modeling real-time systems. In M. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, July 1990.
4. Ch. Baier and J.-P. Katoen. *Principles of Model-Checking*. MIT Press, May 2008.
5. T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer-Verlag, July 2001.

6. G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 312–326, Barcelona, Spain, Mar. 2004. Springer-Verlag.
7. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *Proceedings of the 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, pages 125–126. IEEE Comp. Soc. Press, Sept. 2006.
8. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 2098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2004.
9. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. In R. E. A. Mason, editor, *Information Processing 83 – Proceedings of the 9th IFIP World Computer Congress (WCC'83)*, pages 41–46. North-Holland/IFIP, Sept. 1983.
10. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. C. Kozen, editor, *Proceedings of the 3rd Workshop on Logics of Programs (LOP'81)*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1982.
11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, Sept. 2003.
12. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.
13. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, Jan. 1977.
14. H. Dierks, S. Kupferschmid, and K. G. Larsen. Automatic abstraction refinement for timed automata. In J.-F. Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'07)*, volume 4763 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, Oct. 2007.
15. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems (AVMFSS'89)*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 1990.
16. R. Ehlers, D. Fass, M. Gerke, and H.-J. Peter. Fully symbolic timed model checking using constraint matrix diagrams. In *Proceedings of the 31st IEEE Symposium on Real-Time Systems (RTSS'10)*, pages 360–371. IEEE Comp. Soc. Press, Nov. 2010.
17. R. Ehlers, R. Mattmüller, and H.-J. Peter. Combining symbolic representations for solving timed games. In K. Chatterjee and T. A. Henzinger, editors, *Proceedings of the 8th International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'10)*, volume 6246 of *Lecture Notes in Computer Science*, pages 107–212. Springer-Verlag, Sept. 2010.
18. F. He, H. Zhu, W. N. N. Hung, X. Song, and M. Gu. Compositional abstraction refinement for timed systems. In J. Liu, D. A. Peled, B.-Y. Wang, and F. Wang, editors, *Proceedings of the 4th International Symposium on Theoretical Aspects of Software Engineering (TASE'10)*, pages 168–176. IEEE Comp. Soc. Press, 2010.

19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Conference Record of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, volume 37(1) of *ACM SIGPLAN Notices*, pages 58–70. ACM Press, Jan. 2002.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In T. Ball and S. Rajamani, editors, *Proceedings of the 10th International SPIN Workshop (SPIN'03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, Apr. 2003.
21. F. Herbreteau, D. Kini, B. Srivathsan, and I. Walukiewicz. Using non-convex approximations for efficient analysis of timed automata. In S. Chakraborty and A. Kumar, editors, *Proceedings of the 31st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'11)*, volume 13 of *Leibniz International Proceedings in Informatics*, pages 78–89. Leibniz-Zentrum für Informatik, Dec. 2011.
22. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Lazy abstractions for timed automata. In N. Sharygina and H. Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*, volume 8044 of *Lecture Notes in Computer Science*, pages 990–1005. Springer-Verlag, July 2013.
23. F. Herbreteau and T.-T. Tran. Improving search order for reachability testing in timed automata. In S. Sankaranarayanan and E. Vicario, editors, *Proceedings of the 13th International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'15)*, volume 9268 of *Lecture Notes in Computer Science*, pages 124–139. Springer-Verlag, Sept. 2015.
24. K. L. McMillan. *Symbolic Model Checking — An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1993.
25. T. K. Nguyen, J. Sun, Y. Liu, J. S. Dong, and Y. Liu. Improved BDD-based discrete analysis of timed systems. In D. Giannakopoulou and D. Méry, editors, *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 326–340. Springer-Verlag, Aug. 2012.
26. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57. IEEE Comp. Soc. Press, Oct.-Nov. 1977.
27. S. A. Seshia and R. E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In W. A. Hunt, Jr and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 154–166. Springer-Verlag, July 2003.
28. T. Tóth and I. Majzik. Lazy reachability checking for timed automata using interpolants. In A. Abate and G. Geeraerts, editors, *Proceedings of the 15th International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'17)*, volume 10419 of *Lecture Notes in Computer Science*, pages 264–280. Springer-Verlag, Sept. 2017.
29. F. Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proceedings of the 21st IFIP TC6/WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01)*, volume 197 of *IFIP Conference Proceedings*, pages 235–250. Chapman & Hall, Aug. 2001.
30. W. Wang and L. Jiao. Trace abstraction refinement for timed automata. In F. Cassez and J.-F. Raskin, editors, *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14)*, volume 8837 of *Lecture Notes in Computer Science*, pages 396–410. Springer-Verlag, Nov. 2014.

A Details on Benchmarks

Monoprocess-scheduling benchmarks we considered are listed below. In each case, the `.aag` file is the circuit defining the process (the models with identical file names is available at `http://www.syntcomp.org`). Only a subset of the latches are "clocked", that have time constraints. This can be seen in the file name: for instance, in `amba3b5y.aag_4L_200.xml`, only the first four latches are clocked. The number of clocks is then five (an additional one is used to test the elapsed global time). The last number is the time bound to be tested. The complete list of benchmarks are given in Table 1.

amba3b5y.aag_10L_300.xml
amba3b5y.aag_4L_200.xml
amba3b5y.aag_4L_290.xml
amba3b5y.aag_4L_300.xml
amba3b5y.aag_5L_290.xml
amba3b5y.aag_5L_300.xml
amba3b5y.aag_6L_290.xml
amba3b5y.aag_6L_300.xml
amba3b5y.aag_7L_290.xml
amba3b5y.aag_7L_300.xml
amba3b5y.aag_8L_300.xml
amba3b5y.aag_9L_300.xml
amba4c7y.aag_10L_300.xml
amba4c7y.aag_4L_200.xml
amba4c7y.aag_4L_300.xml
amba4c7y.aag_5L_200.xml
amba4c7y.aag_5L_300.xml
amba4c7y.aag_6L_300.xml
amba4c7y.aag_7L_300.xml
amba4c7y.aag_8L_300.xml
amba4c7y.aag_9L_300.xml
bs16y.aag_4L_100.xml
bs16y.aag_4L_150.xml
bs16y.aag_4L_200.xml
cnt5y.aag_4L_200.xml
cnt5y.aag_4L_300.xml
factory_assembly_3x3_1_1errors.aag_10L_500.xml
factory_assembly_3x3_1_1errors.aag_4L_200.xml
factory_assembly_3x3_1_1errors.aag_4L_300.xml
factory_assembly_3x3_1_1errors.aag_5L_300.xml
factory_assembly_3x3_1_1errors.aag_6L_300.xml

factory_assembly_3x3_1_1errors.aag_6L_400.xml
factory_assembly_3x3_1_1errors.aag_6L_500.xml
factory_assembly_3x3_1_1errors.aag_7L_500.xml
factory_assembly_3x3_1_1errors.aag_8L_500.xml
factory_assembly_3x3_1_1errors.aag_9L_500.xml
genbuf2b3unrealy.aag_10L_400.xml
genbuf2b3unrealy.aag_4L_300.xml
genbuf2b3unrealy.aag_5L_250.xml
genbuf2b3unrealy.aag_5L_300.xml
genbuf2b3unrealy.aag_6L_300.xml
genbuf2b3unrealy.aag_7L_300.xml
genbuf2b3unrealy.aag_7L_400.xml
genbuf2b3unrealy.aag_8L_400.xml
genbuf2b3unrealy.aag_9L_400.xml
genbuf5f5n.aag_10L_300.xml
genbuf5f5n.aag_5L_290.xml
genbuf5f5n.aag_5L_300.xml
genbuf5f5n.aag_6L_290.xml
genbuf5f5n.aag_6L_300.xml
genbuf5f5n.aag_7L_290.xml
genbuf5f5n.aag_7L_300.xml
genbuf5f5n.aag_8L_300.xml
genbuf5f5n.aag_9L_300.xml
moving_obstacle_8x8_1glitches.aag_10L_300.xml
moving_obstacle_8x8_1glitches.aag_4L_150.xml
moving_obstacle_8x8_1glitches.aag_4L_300.xml
moving_obstacle_8x8_1glitches.aag_5L_150.xml
moving_obstacle_8x8_1glitches.aag_5L_300.xml
moving_obstacle_8x8_1glitches.aag_6L_150.xml
moving_obstacle_8x8_1glitches.aag_6L_300.xml
moving_obstacle_8x8_1glitches.aag_7L_150.xml
moving_obstacle_8x8_1glitches.aag_7L_300.xml
moving_obstacle_8x8_1glitches.aag_8L_300.xml
moving_obstacle_8x8_1glitches.aag_9L_300.xml

Table 1: Monoprocess benchmarks

For the multiprocess-scheduling benchmarks, we generated instances using the data shown in Table 2. All models have three clocks, one per process. The first three entries show the circuits (from `http://www.syntcomp.org`) used to define the processes that are being executed. The last number is the number of the scenario, which determines the execution times of arriving tasks according to the value of a selected latch:

– in scenario 0, the two tuples (C_1, C_2, D) of execution time interval and relative deadlines are: $(500, 1000, 3000)$, $(400, 800, 3000)$.
– in scenario 1: $(500, 1000, 1500)$, $(400, 800, 1600)$.
– in scenario 2: $(1000, 1000, 10000)$, $(20000, 20000, 200000)$.

Model	`Process1`	`Process2`	`Process3`	Scenario
0	`amba3b5y.aag`	`add2y.aag`	`add2y.aag`	0
1	`amba3b5y.aag`	`add2y.aag`	`add2y.aag`	1
2	`amba3b5y.aag`	`add2y.aag`	`add2y.aag`	2
3	`cnt4y.aag`	`cnt3y.aag`	`cnt3y.aag`	0
4	`cnt4y.aag`	`cnt3y.aag`	`cnt3y.aag`	1
5	`cnt4y.aag`	`cnt3y.aag`	`cnt3y.aag`	2
6	`cnt4y.aag`	`cnt4y.aag`	`cnt3y.aag`	0
7	`cnt4y.aag`	`cnt4y.aag`	`cnt3y.aag`	1
8	`cnt4y.aag`	`cnt4y.aag`	`cnt3y.aag`	2
9	`cnt4y.aag`	`cnt4y.aag`	`cnt4y.aag`	0
10	`cnt4y.aag`	`cnt4y.aag`	`cnt4y.aag`	1
11	`cnt4y.aag`	`cnt4y.aag`	`cnt4y.aag`	2
12	`cnt5y.aag`	`cnt4y.aag`	`cnt3y.aag`	0
13	`amba3b5y.aag`	`cnt3y.aag`	`cnt3y.aag`	2
14	`cnt5y.aag`	`cnt3y.aag`	`cnt3y.aag`	0
15	`cnt5y.aag`	`cnt3y.aag`	`cnt3y.aag`	1
16	`cnt5y.aag`	`cnt3y.aag`	`cnt3y.aag`	2
17	`cnt3y.aag`	`cnt3y.aag`	`cnt3y.aag`	0
18	`cnt3y.aag`	`cnt3y.aag`	`cnt3y.aag`	1
19	`cnt3y.aag`	`cnt3y.aag`	`cnt3y.aag`	2
20	`amba3b5y.aag`	`add2y.aag`	`add2y.aag`	1
21	`amba3b5y.aag`	`add2y.aag`	`add2y.aag`	0
22	`amba3b5y.aag`	`add2y.aag`	`cnt3y.aag`	1
23	`amba3b5y.aag`	`add2y.aag`	`cnt3y.aag`	0
24	`bs8y.aag`	`add2y.aag`	`add2y.aag`	0
25	`bs8y.aag`	`add2y.aag`	`add2y.aag`	1
26	`bs8y.aag`	`add2y.aag`	`add2y.aag`	2
27	`bs8y.aag`	`bs8y.aag`	`add2y.aag`	0
28	`bs8y.aag`	`bs8y.aag`	`add2y.aag`	1
29	`bs8y.aag`	`bs8y.aag`	`add2y.aag`	2
30	`bs8y.aag`	`bs8y.aag`	`bs8y.aag`	0

Model	`Process1`	`Process2`	`Process3`	Scenario
31	`bs8y.aag`	`bs8y.aag`	`bs8y.aag`	1
32	`bs8y.aag`	`bs8y.aag`	`bs8y.aag`	2
33	`mv4y.aag`	`mv4y.aag`	`add2y.aag`	0
34	`mv4y.aag`	`mv4y.aag`	`add2y.aag`	1
35	`mv4y.aag`	`mv4y.aag`	`add2y.aag`	2
36	`mv4y.aag`	`mv4y.aag`	`mv4y.aag`	2
37	`stay2y.aag`	`stay2y.aag`	`mv4y.aag`	2
38	`stay4y.aag`	`add2y.aag`	`add2y.aag`	2
39	`stay4y.aag`	`cnt4y.aag`	`add2y.aag`	2
40	`stay4y.aag`	`stay2y.aag`	`mv4y.aag`	2
41	`stay4y.aag`	`stay2y.aag`	`stay2y.aag`	2

Table 2: Multiprocess benchmarks

Information on asynchronous-computation benchmarks is listed in Table 3. The number of clocks in each model is equal to the number of non-input gates.

`File name`	Number of gates	Number of inputs	Time bound
`a0`	8	4	50
`a1`	8	4	150
`a2`	9	4	50
`a3`	9	4	150
`a4`	9	4	400
`a5`	16	8	50
`a6`	16	8	150
`a7`	19	14	150
`a8`	19	14	300
`a9`	20	14	300
`a10`	20	14	300
`b0`	9	4	1000
`b1`	10	4	1000
`b2`	9	4	1000
`b3`	16	8	1000
`b4`	40	35	1000
`b5`	20	14	1000
`b6`	20	15	1000
`b7`	9	4	1000
`b8`	10	3	1000
`b9`	19	14	1000

`File name`	Number of gates	Number of inputs	Time bound
`b10`	16	8	1000

Table 3: Asynchronous Computation Benchmarks