Culling predicates for the Verification of Real-Time Models

Technical Report soft-08-06
Chair for Software Engineering

University of Konstanz

Culling predicates for the Verification of Real-Time Models

Bahareh Badban

Department of Software Engineering
University of Konstanz, Germany

Abstract. We present an algorithm that generates invariants for real-time models. The algorithm, further, prunes the model by first detecting, and then removing idle discrete transitions (transitions which can never be traversed). We next demonstrate how the generated invariants can be used to create a finite-state abstraction for the original model. To this end, we enhance the idea of predicate abstraction through fully incorporating locations of the concrete timed automata model in the abstraction phase.

1 Introduction

Predicate abstraction, an instance of the general theory of abstract interpretation [5], is a technique for generating finite (abstract) models of finite/infinite state systems. This technique was first introduced by Graf and Saïdi [8] as a method for automatically determining invariant properties of infinite-state systems. This approach recently has attracted a lot of research in software verification projects. To this end, lots of methods for the verification of large (finite-state) models have been introduced. The idea is that based on a given finite set of predicates, which could be provided by the user, a finite state abstraction of the system is generated. This abstraction has at most $2^{\|P\|}$ states, where $\|P\|$ is the number of predicates [14].

Predicate abstraction involves approximating a concrete transition system (or a set of concrete states) using a set of formulas, P, called *predicates*. These predicates usually denote some properties of the states. The abstraction is defined by the value (true or false) of these predicates, in any concrete state of the system [16].

In this paper, we provide an automatic approach for generating *invariants* in real-time models. We use these invariants as the initial set of predicates to abstract the original model. The major difference between our work and previous works on predicate abstraction is two-fold: we provide a method for generating predicates in real time systems, where existing methods either do not involve time, or do not provide a method for obtaining the set of predicates, since this set is considered to be provided by the user, or is generated based on trial and error approach. Second, we introduce a new way of abstraction, where states are not only made of combinations of (negated) predicates, but also they contain the concrete states (in their structure), where each concrete state is coupled with its invariant. To our knowledge, this is the first time that abstraction is built this way. Our main motivation for this article is nicely explained by Das and Dill in [6]:

Another problem is how to discover the appropriate set of predicates. In much of the work on predicate abstraction, the predicates were assumed to be given by the user, or they were extracted syntactically from the system description. It is obviously difficult for the user to find the right set of predicates (indeed, it is trial-and-error process involving inspecting failed proofs), and the predicates appearing in the system description are rarely sufficient. There has

been less work and less progress, on solving the problem of finding the right set of predicates. In addition... there is a challenge of avoiding irrelevant predicates...

Related Work. Colón and Uribe, in [4], apply technique of predicate abstraction to make weakly preserving abstraction of transition systems. Using a set of predicates called, *basis*, they provide an automatic approach to entail under and over-approximation of the real time systems. However, as said in the paper itself, the choice of abstraction basis can be based on the user's understanding of the system, and that techniques for the (manual/automatic) generation of the abstraction basis remains to be tested and explored. A similar approach with extra method on refinement is later introduced in [19] by Möller, et.al. The same disadvantage remains in this work too.

Das and Dill, use predicate abstraction approach in [6], where they use spurious traces to discover predicates. Their other work on predicate abstraction is [7]. In [11] Hoffmann, et.al. use the technique of predicate abstraction to abstract timed-models and then to find good heuristics. Their intention is to use heuristic guided search to check safety properties. In [16], Lahiri, et.al. use SMT for predicate abstraction. The problem there, is to have a fast approach for approximating a set of concrete states using a set of predicates.

Podelski and Rybachenko, extend the predicate abstraction technique to transition-predicate-abstraction in [20]. The method is based on checking termination under fairness, which is suitable for checking liveness properties. Their models do not involve time.

Jhala and McMillan in [12] also Henzinger, et.al. in [9] use intrpolation to prove whether an abstract trace is infeasible, and also to extract predicates from the proof. Then, in the refinement phase, they extract the infeasible trace. In [2] Ball, et.al. introduce an abstraction method based on oracle-guided widening. In most cases the widening is such that it simply drops a variable from the rest of the computations of the pre-states. Their approach, as they mention, works perfectly for the pre-processing method, in a sense that whenever the abstractions succeeds (i.e. it is safe) then the concrete model will succeed too. The approach seeks a breakthrough with regard to post-processing methods.

Symbolic approach is also used in the predicate abstraction domain to reduce the number of calls to the theorem-prover. Examples of this are [13, 15].

The paper is organized as follows. Section 2 provides some preliminary definitions on real-time automata. Semantics of timed automata and their possible transitions are discussed in section 2.1. Section 3 introduces our method of creating new invariants. The related algorithm, called CIPM, is explained in Section 3.1. In Section 3.2 we exemplify our algorithm by an example which is used in [19]. Technique of predicate abstraction and our approach on that, are described in Sections 4 and 4.1. In Sections 4.1 we continue the example of Section 3.2 to build abstraction for the original model. Finally, Section 5 presents some conclusions and future works.

2 Timed Automata

A *timed automaton*, consists of a finite automaton which describes the system *control* states and its transitions, and a finite set of clocks which keep track of the *time* elapsed since the last reset [1, 3]. *Clocks* are non-negative real valued variables. Initially, all clocks are assigned to 0, then they all evolve at the same speed, synchronously with time. *Configurations* of the system are given by the current control location of the automata and the value of each clock, denoted $\langle l, u \rangle$, where l is the control location, and u is the valuation function which assigns

to each clock its current value. $u + d$, for $d \in \mathbb{R}^+$, is a valuation which assigns to each clock x, $u(x) + d$, i.e. it increases the value of all clocks by d.

For a set X of clock variables, $\mathsf{G}(X)$, the set of *clock constraints* g is defined by:

$$g := x \leq t \mid t \leq x \mid \neg g \mid g_1 \wedge g_2$$

where $x \in X$, and t, called *term*, is either a variable in X or a constant in $\mathbb{Q}$. By $var(g)$ we denote the set of clock variables occurring in g. Timed automata is formally defined as:

Definition 1. A timed automaton $\mathcal{A}$ is a tuple $\langle L, l_0, \Sigma, X, I, E \rangle$ where

- L is a finite set of locations (or states), called *control locations*,
- $l_0 \in L$ is the initial location,
- Σ is a finite set of labels, called *events*,
- X is a finite set of clocks,
- $\mathcal{I} : L \mapsto \mathsf{G}(X)$ assigns to each location in L some clock constraint in $\mathsf{G}(X)$,
- $E : L \times \Sigma \times 2^X \times \mathsf{G}(X) \longrightarrow L$ represents *discrete* transitions.

The clock constraint associated to each location $l \in L$, is called its *invariant*, denoted $\mathcal{I}(l)$, and it requires that time can flow in a location only as long as its invariant remains true. So, $\mathcal{I}(l)$ must hold when the current state is l.

We call a constraint g *atomic* if it is equivalent to $s \leq t,\ t \leq s$ or their negation, for terms s and t. To each clock constraint g, we associate a set of its atomic sub-formulas, denoted $\mathsf{atom}(g)$, defined as:

- $\mathsf{atom}(g) := \{g\}$ when g is atomic,
- $\mathsf{atom}(g_1 \wedge g_2) := \mathsf{atom}(g_1) \cup \mathsf{atom}(g_2)$.

Reset constraints, denoted as r, are conjunction of (one or more) formulas of the form $x := c$ where c is a constant in $\mathbb{Q}$. We similarly apply atom over reset constraints as well, e.g. $\mathsf{atom}(x := 3 \wedge y := 0) \equiv \{x := 3,\ y := 0\}$. An atomic constraint is called *bounded* if it is of the form $x \prec c$, where c is a constant value and $\prec \in \{=, <, \leq\}$. g is *unbounded* if it is not bounded. For example, $x \leq y$ and $x > 2$ are unbounded. Based on this, we define $^{\mathsf{un}}(A) := \{a \in A \mid a \text{ is unbounded}\}$.

We say a valuation u *satisfies* g, denoted $u \models g$, if the assigned value to all variables in g by u, satisfies g. For example if $g := x + 1 > y \wedge x < 5$, and $u(x) = 4.1$ and $u(y) = 3$, then $u \models g$. We consider true as a valid proposition which is satisfied by each valuation, i.e. $u \models \mathsf{true}$ for each u. $u \models A$, for a set A, if $u \models a$ for each $a \in A$. Given two constraints g and g', g entails g', denoted $g \Rightarrow g'$, if whenever $u \models g$ then $u \models g'$, for any valuation u. For instance, $x \geq 3 \Rightarrow x > 2$. Based on this, we define a function max as:

$$\mathsf{max}(g, g') := \begin{cases} g' & \text{if } g \Rightarrow g' \\ g & \text{if } g' \Rightarrow g \\ \mathsf{true} & \text{if } \neg g \Rightarrow g' \text{ or equivalently, } \neg g' \Rightarrow g \\ g \vee g' & \text{otherwise} \end{cases}$$

This definition is further extended over sets:

$$\mathsf{max}(A, B) := \bigvee_{(a,b) \in \mathsf{A}|\mathsf{B}} \mathsf{max}(a, b)$$

where $\mathsf{A}|\mathsf{B} := \{(a,b) \mid a \in A,\ b \in B \text{ and } var(a) = var(b)\}$. For example, if $A = \{x < y, x > 2\}$ and $B = \{x < 3\}$ then $\mathsf{A}|\mathsf{B} = \{(x > 2, x < 3)\}$ and hence, $\mathsf{max}(\{x < y, x > 2\}, \{x < 3\}) = \mathsf{max}(x > 2, x < 3) = \mathsf{true}$, since $x \geq 3 \Rightarrow x > 2$.

In each timed automaton, the value of all clocks evolve with the same speed (of slope 1) with time. Meaning that if at time t_0, x has the value of x_0 then after Δt time units the value of x will be $x_0 + \Delta t$. This, brings about the next property:

Note 1 In a timed automaton $\mathcal{A}$, if at some point of time (e.g. t_0) a relation like $x \prec y$, holds between two clock variables x and y where $\prec \in \{=, <, \leq\}$, then this relation will be preserved also in the next time units until one of the variables is reset (to a new constant value). This is because $x(t) = x + \Delta t \prec y + \Delta t = y(t)$, where Δt computes the time elapsed as of t_0.

Lemma 1. *If $u \models g$ for a valuation u and some* unbounded *atomic constraint g, then $u+d \models g$ for all $d \in \mathbb{R}^+$. This can be extended to any set of unbounded atomic constraints, too.*

Proof. If g is of the form $x \geq c$ or $x > c$, then proof is obvious. For other constraints g, by Note 1, $u + d \models g$ (d is the Δt). Obviously, by definition, this property extends to sets of unbounded atomic constraints, too. ■

Lemma 2. *For each two atomic constraints g and g', and sets A and B of atomic constraints, we have:*

- $g \Rightarrow \mathsf{max}(g, g')$ *and* $g' \Rightarrow \mathsf{max}(g, g')$.
- *if $u \models g$ for some valuation u, then $u \models \mathsf{max}(g, g')$.*
- *if $u \models A$ (or $u \models B$) for some valuation u, then $u \models \mathsf{max}(A, B)$.*

Proof. – This is obvious according to the definition of max.
- By definition, this is equivalent to the first item.
- $u \models A$, hence by definition, $u \models a$ for all $a \in A$. Therefore, according to the second item, for all $a \in A$ and $b \in B$, $u \models \mathsf{max}(a, b)$. Hence, by definition, $u \models \mathsf{max}(A, B)$. This holds, analogously, for when $u \models B$. ■

2.1 Semantics of timed automata

To each timed automaton $\mathcal{A}$, [1] a transition system $\mathcal{S}_\mathcal{A}$ is associated. States of $\mathcal{S}_\mathcal{A}$ are pairs $\langle l, u\rangle$, where $l \in L$ is a control location of $\mathcal{A}$ and u is a valuation over X which satisfies $\mathcal{I}(l)$, i.e. $u \models \mathcal{I}(l)$. $\langle l_0, u\rangle$ is an *initial* state of $\mathcal{S}_\mathcal{A}$ if l_0 is the initial location of $\mathcal{A}$ and for all $x \in X$: $u(x) = 0$.

Transitions. For each transition system $\mathcal{S}_\mathcal{A}$, the system configuration changes by two kinds of transitions:

- a *delay transition*, by letting time delay $d \in \mathbb{R}^+$ elapse. The value of all clocks would then increase by d, leading to the transition: $\langle l, u\rangle \overset{d}{\mapsto} \langle l, u + d\rangle$.
- a *discrete transition*, by enabling[2] a transition. In this case all clocks, except those which are reset, remain unchanged. This leads to the transition $\tau := \langle l, u\rangle \overset{a,g,r}{\mapsto} \langle l', u'\rangle$ where a is an event, g is a clock constraint and r is a reset constraint.

[1] We work with nondeterministic timed automaton.
[2] A transitions is *enabled* if it can be taken at a given state.

An *execution* of a system is a (possibly infinite) sequence of configurations, $\langle l, u\rangle$, where two successive configurations correspond to either a discrete or a delay transition.

In the sequel, τ and d denote discrete and delay transitions, respectively. We may denote a discrete transition τ as $\langle l, u\rangle \overset{\tau}{\mapsto} \langle l', u'\rangle$ when a, g, r are not necessary to be clarified. For each $\tau := \langle l, u\rangle \overset{a,g,r}{\mapsto} \langle l', u'\rangle$, we define $\mathsf{G}_\tau := \mathsf{atom}(g)$ and $\mathsf{R}_\tau := \mathsf{atom}(r)$. For this transition, l and l' are, respectively, called source of τ, denoted $\mathsf{sorc}(\tau)$, and target of τ, denoted $\mathsf{tar}(\tau)$. $\mathsf{G}_{\tau/\mathsf{R}_\tau}$ (resp. $\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}$) represents the set of all atomic constraints in G_τ (resp. $\mathcal{I}(\mathsf{sorc}(\tau))$) which do not have a variable occurring in R_τ. For instance, if $\tau := \langle l, u\rangle \overset{x\le y\wedge z<x+1,\ z:=0}{\mapsto} \langle l', u'\rangle$, then $\mathsf{G}_\tau = \{x \le y,\ z < x+1\}$, $\mathsf{R}_\tau = \{z := 0\}$ and $\mathsf{G}_{\tau/\mathsf{R}_\tau} = \{x \le y\}$. In this example, z occurs in R_τ. For this transition if $\mathcal{I}(l) = \{y < 2, z > 4\}$ then $\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau} = \{y < 2\}$.

For each discrete transition τ we define:

$$\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \ \cup\ {}^{\mathsf{un}}(\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \ \cup\ \overline{\mathsf{atom}}(\mathsf{R}_\tau)$$

where $\overline{\mathsf{atom}}(\mathsf{R}_\tau) = \bigcup_{r\in\mathsf{atom}(\mathsf{R}_\tau)} \overline{r}$, and

$$\overline{x := c} = \begin{cases} \{x \le y \mid y \in X - \{x\}\} & \text{if } c = 0 \\ \{x \ge c\} \cup \{x \le y + c \mid y \in X - \{x\}\} & \text{if } c > 0 \end{cases}$$

We will later show (Theorem 3) that $\mathsf{inv}(\tau)$ is a set of constraints that is preserved in $\mathsf{tar}(\tau)$. Below, in the next lemma, we prove this for when we have just entered a state (before any time elapse). For the example above, $\tau := \langle l, u\rangle \overset{x\le y\wedge z<x+1,\ z:=0}{\mapsto} \langle l', u'\rangle$, $\mathsf{inv}(\tau) = {}^{\mathsf{un}}(\{x \le y\}) \cup {}^{\mathsf{un}}(\{y < 2\}) \cup \overline{z := 0} = \{x \le y\} \cup \emptyset \cup \{z \le x, z \le y\} = \{x \le y, z \le x, z \le y\}$. Here, $X = \{x, y, z\}$.

Lemma 3. *For any discrete transition $\langle l, u\rangle \overset{\tau}{\mapsto} \langle l', u'\rangle$, we have $u' \models \mathsf{inv}(\tau)$.*

Proof. Clock variables are reset iff they occur in R_τ. Hence, those variables which do not occur in R_τ retain their value when the transition τ is taken place. These are variables which occur in $\mathsf{G}_{\tau/\mathsf{R}_\tau}$. Therefore, $u' \models \mathsf{G}_{\tau/\mathsf{R}_\tau}$, and since $\mathsf{G}_{\tau/\mathsf{R}_\tau} \subseteq {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau})$, $u' \models {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau})$. The same reason also works for ${}^{\mathsf{un}}(\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau})$. Now, since u' has some new values for the reset variables in r, hence $u' \models \overline{r}$ right at the time of transition (since, time has not elapsed yet). Therefore, $u' \models \overline{\mathsf{atom}}(\mathsf{R}_\tau)$. Summing these up results in $u' \models \mathsf{inv}(\tau)$. ■

Definition 2. For each control location l, we define a set of incoming discrete transitions, ${}^{\mathsf{in}}\mathsf{trans}(l, \mathcal{A})$, and a set of outgoing discrete transitions, ${}^{\mathsf{out}}\mathsf{trans}(l, \mathcal{A})$:

$$\begin{aligned} {}^{\mathsf{in}}\mathsf{trans}(l, \mathcal{A}) &:= \{\tau \mid \exists l_i, u_i, u :\ \langle l_i, u_i\rangle \overset{\tau}{\mapsto} \langle l, u\rangle\} \\ {}^{\mathsf{out}}\mathsf{trans}(l, \mathcal{A}) &:= \{\tau \mid \exists l', u', u :\ \langle l, u\rangle \overset{\tau}{\mapsto} \langle l', u'\rangle\} \end{aligned}$$

Notice that in this definition we only consider discrete transitions.

Below, we define a reduction system. This system simplifies (disjunction of) clock constraints. The idea is that if some valuation function u satisfies the left hand-side of the reduction step (denoted by $\longrightarrow$), then it will satisfy the right-hand side too.

Definition 3 (Reduction System). We apply the following reduction rules on disjunction of constraints, ϕ. Here, s and t are terms.

1. $s < t \vee s = t \longrightarrow s \leq t$
2. $s < t \vee s > t \longrightarrow s \neq t$

ϕ is called *simplified* if none of the rules above can be applied on it. We apply the *reduction rules* of Definition 3 on ϕ as long as they can be applied (i.e. result is not identical to the constraint itself). When reduction process terminates, we call the result $\mathsf{simp}(\phi)$. Obviously, $\mathsf{simp}(\phi)$ is simplified, since by definition no rule can be applied on it.

Lemma 4. *If $u \models \phi$ then $u \models \mathsf{simp}(\phi)$, for each valuation u.*

Proof. If $\phi \longrightarrow \psi$ with any of the reduction rules, then obviously $u \models \psi$. Therefore, $u \models \mathsf{simp}(\phi)$ too. ■

Next section, introduces an automatic approach to create new invariants in the control locations, and to further prune the original model, by removing the redundant transitions. These are the transitions which can never be traversed, and wherefore do not impact the reachability analysis in the model.

3 Creating new invariants

In this section we explain the algorithm, which we use to generate new invariants in the control locations. In other-words, we strengthen the already given (original) invariants in each control location. Based on this new set of invariants, we will later introduce a set of predicates, P. This set will be used to abstract the original model, using the predicate abstraction technique, cf. Section 4.

3.1 The Algorithm

Algorithm 1, called CIPM, introduces a method for creating new invariants at each control location of the input automaton. The input of this algorithm is a timed automaton. We assume that each location is given a separate index from 0 to $\|\mathcal{A}\| - 1$, e.g. l_1, where $\|\mathcal{A}\|$ is the number of control locations in $\mathcal{A}$. Before, explaining the behaviour of the algorithm, we need to introduce some fundamental definition:

Definition 4. A discrete transition τ is called *idle* if it is never enabled. This, for instance can happen when the constraint over the transition is never being satisfied, or the valuation function, obtained from the transition, does not satisfy the invariant of the target location.

For example if τ is the discrete transition $\langle l, u\rangle \overset{x \leq y}{\mapsto} \langle l', u'\rangle$, where $x > y + 1$ is invariant in location l, i.e. $\mathcal{I}(l) = \{x > y + 1\}$, then this transition is idle, since the constraint $x \leq y$ is never going to be satisfied as long as we are in l.

The CIPM algorithm, first collects all the original invariants in each location l_i, denoted $\mathcal{I}(l_i)$. Then, selects each location l_i, at a time, and collects its incoming transitions in ${}^{\mathsf{in}}\mathsf{trans}(l_i, \mathcal{A})$. With respect to Lemma 5, the idle transitions are then detected, and deleted from the model. Meanwhile, $\mathsf{inv}(\tau)$ is computed for non-idle transitions to extract the constraints that are transformed into l_i via τ. Applying max over all these constraints entails, $\mathsf{inv}(l_i)$, the final constraint imposed on l_i by all transitions in ${}^{\mathsf{in}}\mathsf{trans}(l_i, \mathcal{A})$. Since, l_i might also have some original invariant, $\mathcal{I}(l_i)$, the final constraint over valuations in l_i will be the conjunction of these two. This is expressed by $\mathcal{I}_{\mathcal{A}}(l_i) := \mathcal{I}(l_i) \wedge \mathsf{simp}(\mathsf{inv}(l_i))$. Next, having new

invariants in l_i, the outgoing transitions are checked one by one for being idle, using Lemma 5 (1st item). The idle transitions are removed from the model. seen is a set which stores the traversed transitions. This set makes sure that all transitions are (only) once checked for being idle.

In algorithm 1, we use conjunction of sets, which is defined as: $A \wedge B := \bigwedge_{1\leq i\leq n} a_i \wedge \bigwedge_{1\leq j\leq m} b_j$ for $A = \{a_1, ...a_n\}$ and $B = \{b_1, ...b_m\}$. There, $A \wedge \emptyset = \emptyset \wedge A := \{\mathsf{true}\} \wedge A$.

Lemma 5. *A discrete transition τ is idle, if one of the conditions below, holds:*

- $\mathcal{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ *is a contradiction,*
- $\mathsf{inv}(\tau) \wedge \mathcal{I}(\mathsf{tar}(\tau))$ *is a contradiction.*

Proof. – $\mathcal{I}(\mathsf{sorc}(\tau))$ holds as long as the current location is $\mathsf{sorc}(\tau)$. At this location, τ is enabled only when G_τ holds. If this occurs then $\mathcal{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ holds. This, by assumption, can never happen.
– By Lemma 3, if $\langle l, u\rangle \overset{\tau}{\mapsto} \langle l_i, u_i\rangle$ is enabled then $u_i \models \mathsf{inv}(\tau)$. By definition, $u_i \models \mathcal{I}(\mathsf{tar}(\tau))$ too. Therefore, $u_i \models \mathsf{inv}(\tau) \wedge \mathcal{I}(\mathsf{tar}(\tau))$. This contradicts the assumption. So, τ is never enabled. ■

We say two timed automata $\mathcal{A}$ and $\mathcal{A}_1$ are *equivalent*, denoted $\mathcal{A} \doteq \mathcal{A}_1$, if they differ only on some idle transitions.

Theorem 1. *The* CIPM *algorithm has the following properties:*

- *it is terminating.*
- *if* $\mathsf{CIPM}(\mathcal{A}_1) = (\mathcal{A}, \mathcal{I}_\mathcal{A})$ *then* $\mathcal{A} \doteq \mathcal{A}_1$.
- *for each control location l,* $\mathsf{inv}(l)$ *consists of unbounded constraints (only).*

Proof. – In timed automata, number of control locations ($\|\mathcal{A}\|$) is a finite number, say n. So, the first two (**repeat**) loops halt after n steps. Besides, each timed automaton consists of a finite automaton which describes the system control states and its transitions. Hence, each control location has only a finite number of incoming and outgoing discrete transitions. Thus, the other two (**while**) loops will also stop after finite steps.
– According to Lemma 5, and the algorithm, $\mathcal{A}$ is updated only by removing some idle transitions. Therefore, the output automaton will be either the exact same automaton, or it will be an automaton with only less number of idle transitions.
– This can be easily derived because of the definition of $inv(\tau)$ and the definition of $\mathsf{inv}(l)$ in the algorithm. ■

Note 2 Since according to the theorem above, $\mathcal{A} \doteq \mathcal{A}_1$, in the sequel we may use A and A_1 interchangeably.

The new constraint $\mathcal{I}_\mathcal{A}(l)$ implies the original invariant $\mathcal{I}(l)$ and moreover it extracts a stronger clock constraint which should hold as long as we stay in l. We prove this in the next two theorems.

Theorem 2. *If* $\mathsf{CIPM}(\mathcal{A}_1) = (\mathcal{A}, \mathcal{I}_\mathcal{A})$*, then* $\mathcal{I}_\mathcal{A}(l) \Rightarrow \mathcal{I}(l)$ *for each control location l in $\mathcal{A}$, where $\mathcal{I}(l)$ is the original invariant of l in A_1.*

Algorithm 1 Creating Invariants and Pruning the Model (CIPM)

REQUIRES: a timed automaton $\mathcal{A}$

$n := \|\mathcal{A}\|$ %% the number of control locations in $\mathcal{A}$
$X :=$ the set of all clock variables occurred in $\mathcal{A}$
$i := 0$
$j := 0$
$\mathsf{seen} := \emptyset$
repeat
  $\mathcal{I}(l_j) :=$ the given (original) invariant of l_j
  $j := j + 1$
until $j < n$
repeat
  if $i = 0$ **then**
    $\mathsf{inv}(l_i) := \{x = y \mid x, y \in X$ where x and y are not identical$\}$
  else
    $\mathsf{inv}(l_i) := \emptyset$
  $k := 0$
  $\mathsf{In} := {}^{\mathsf{in}}\mathsf{trans}(l_i, \mathcal{A})$
  if $\mathsf{In} = \emptyset \wedge i > 0$ **then**
    $\mathcal{A} := \mathcal{A} \backslash {}^{\mathsf{out}}\mathsf{trans}(l_i, \mathcal{A})$
  else
    while $\mathsf{In} \neq \emptyset \ \wedge \ (k = 0 \vee \mathsf{inv}(l_i) \neq \emptyset)$ **do**
      choose $\tau \in \mathsf{In}$
      $\mathsf{In} := \mathsf{In} \backslash \{\tau\}$
      if $\tau \notin \mathsf{seen}$ **then**
        if $\mathcal{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ is a contradiction **then**
          $\mathcal{A} := \mathcal{A} \backslash \{\tau\}$ %% the idle transition
        else
          $\mathsf{seen} := \mathsf{seen} \cup \{\tau\}$
      $\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \ \cup \ {}^{\mathsf{un}}(\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \ \cup \ \overline{\mathsf{atom}}(\mathsf{R}_\tau)$
      if $\mathsf{inv}(\tau) \wedge \mathcal{I}(l_i)$ is a contradiction **then**
        $\mathcal{A} := \mathcal{A} \backslash \{\tau\}$ %% the idle transition
      else
        $k := k + 1$
        if $k = 1$ **then**
          $\mathsf{inv}(l_i) := \mathsf{inv}(\tau)$
        else
          $\mathsf{inv}(l_i) := \mathsf{max}(\mathsf{inv}(l_i), \mathsf{inv}(\tau))$
    $\mathcal{I}(l_i) := \mathcal{I}(l_i) \wedge \mathsf{simp}(\mathsf{inv}(l_i))$
    $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_i)$
    while $\mathsf{Out} \not\equiv \emptyset$ **do**
      choose $\tau \in \mathsf{Out}$
      $\mathsf{Out} := \mathsf{Out} \backslash \{\tau\}$
      if $\tau \notin \mathsf{seen}$ **then**
        if $\mathcal{I}(l_i) \wedge \mathsf{G}_\tau$ is a contradiction **then**
          $\mathcal{A} := \mathcal{A} \backslash \{\tau\}$ %% the idle transition
        else
          $\mathsf{seen} := \mathsf{seen} \cup \{\tau\}$
  $i := i + 1$
until $i < n$
$\mathcal{I}_\mathcal{A} := \{l_i \mapsto \mathcal{I}(l_i)\}$
return $(\mathcal{A}, \mathcal{I}_\mathcal{A})$

Proof. $\mathcal{I}_{\mathcal{A}}(l) = \mathcal{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$. According to the definition of $\Rightarrow$, we need to show that if $u \models \mathcal{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$ then $u \models \mathcal{I}(l)$, for any valuation function u. Let $\mathcal{I}(l) = \{a_1, ...a_n\}$ and $\mathsf{simp}(\mathsf{inv}(l)) = \{b_1, ...b_m\}$. Therefore, $u \models \bigwedge_{1 \leq i \leq n} a_i \wedge \bigwedge_{1 \leq j \leq m} b_j$, and hence, for each $1 \leq i \leq n$, $u \models a_i$. By definition, this means that $u \models \mathcal{I}(l)$. ■

Next theorem, shows that $\mathcal{I}_{\mathcal{A}}$ associates to each control location l a set of new invariants.

Theorem 3. *If* $\mathsf{CIPM}(\mathcal{A}_1) = (\mathcal{A}, \mathcal{I}_{\mathcal{A}})$, *then* $u \models \mathcal{I}_{\mathcal{A}}(l)$, *for each configuration* $\langle l, u \rangle$ *in* $\mathcal{S}_{\mathcal{A}1}$. *In other words,* $\mathcal{I}_{\mathcal{A}}(l)$ *is invariant in* l.

Proof. Since $\mathcal{I}_{\mathcal{A}}(l) = \mathcal{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$, we need to prove that $u \models \mathcal{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$ where $\mathcal{I}(l)$ is the original (given) invariant of location l (cf. Definition 1). To this end, we show that $u \models \mathcal{I}(l)$ and $u \models \mathsf{simp}(\mathsf{inv}(l))$. First part holds by definition. For the second part, by Lemma 4, we only need to prove that $u \models \mathsf{inv}(l)$. We split the proof into two, when $\langle l, u \rangle$ is reached by a discrete transition τ and when it is reached by a delay transition d.

- Assume that $\langle l, u \rangle$ is reached by a discrete transition τ (i.e. $... \overset{\tau}{\mapsto} \langle l, u \rangle$). Then $u \models \mathsf{inv}(\tau)$ by Lemma 3. Hence, $u \models \mathsf{max}(\mathsf{inv}(l), \mathsf{inv}(\tau))$ by Lemma 2 (2nd item). This, according to the algorithm, is the updated value of $\mathsf{inv}(l)$. Therefore, $u \models \mathsf{inv}(l)$.
- Assume that $\langle l, u \rangle$ is reached by a delay transition d. Then, there exist a discrete transition τ, an valuation u_1 and a delay value $d_1 \in \mathbb{R}^+$ such that $... \overset{\tau}{\mapsto} \langle l, u_1 \rangle \overset{d_1}{\mapsto} \langle l, u \rangle$, i.e. $u := u_1 + d_1$ and $\langle l, u_1 \rangle$ is reached by the discrete transition τ. Therefore, according to the previous part we get $u_1 \models \mathsf{inv}(l)$. According to Theorem 1 (last item), all elements of $\mathsf{inv}(l)$ are unbounded, hence, $u_1 + d_1 \models \mathsf{inv}(l)$ by Lemma 1. This completes the proof. ■

3.2 Example

Here, we exemplify our CIPM algorithm, using the example in [19]. Figure 1 shows an example of a timed automaton, $\mathcal{A}$. In this model, x and y are clock variables. l_0, l_1 and l_2 are control

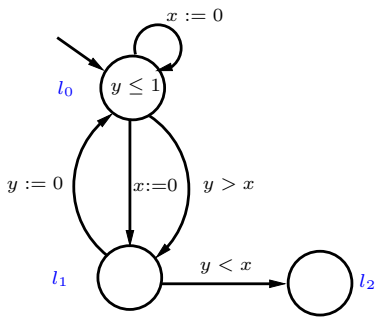

Fig. 1. $\mathcal{A}$. In this timed automaton, x and y are clock variables.

locations. Our intention is to prune this automaton, and to find a new set of invariants in each location. The model starts with the initial value of $x = y = 0$ in location l_0. $y \leq 1$ is the original invariant in l_0, i.e. $\mathcal{I}(l_0) = \{y \leq 1\}$. So, we can stay in l_0 only so long as the value of y does not exceed 1. Once this value has passed 1 then one of the outgoing transitions must be (enabled and) taken out of this state (e.g. $l_0 \overset{x:=0}{\mapsto} l_1$). For the other locations we have: $\mathcal{I}(l_1) = \mathcal{I}(l_2) = \emptyset$.

Now we apply the CIPM algorithm on this automaton:
Initially, $n := 3$, $X := \{x, y\}$, $i := 0$, $j := 0$ and $\mathsf{seen} := \emptyset$. In the first **repeat** loop, for $0 \leq j < 3$ the algorithm collects the original invariants at l_j. So, $\mathcal{I}(l_0) = \{y \leq 1\}$ and $\mathcal{I}(l_1) = \mathcal{I}(l_2) = \emptyset$. In the second **repeat** loop, we get: $\mathsf{inv}(l_0) := \{x = y\}$, $k := 0$, and $\mathsf{In} := {}^{\mathsf{in}}\mathsf{trans}(l_0, \mathcal{A}) = \{l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. Since the condition of the first **if** loop does not hold, the **while** loop must be activated. Here, $\mathsf{In} \neq \emptyset \ \wedge\ (k = 0 \vee \mathsf{inv}(l_0) \neq \emptyset)$ holds, we choose $\tau := l_0 \overset{x:=0}{\mapsto} l_0$ from In, and let $\mathsf{In} := \{l_1 \overset{y:=0}{\mapsto} l_0\}$. $\tau \notin \mathsf{seen}$ (which is $\emptyset$) and $\mathcal{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau = \{y \leq 1\} \wedge \emptyset$, this by definition is $y \leq 1$, which is not a contradiction. Hence, $\mathsf{seen} := \mathsf{seen} \cup \{\tau\} = \{l_0 \overset{x:=0}{\mapsto} l_0\}$. Now, since for this τ we have: ${}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) = \emptyset$, ${}^{\mathsf{un}}(\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) = {}^{\mathsf{un}}(\{y \leq 1\}) = \emptyset$ and $\overline{\mathsf{atom}}(\mathsf{R}_\tau) = \{x \leq y\}$, we derive: $\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \ \cup\ {}^{\mathsf{un}}(\mathcal{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \ \cup\ \overline{\mathsf{atom}}(\mathsf{R}_\tau) = \{x \leq y\}$. Then, $\mathsf{inv}(\tau) \wedge \mathcal{I}(l_0) = \{x \leq y\} \wedge \{y \leq 1\} = x \leq y \wedge y \leq 1$ which is not a contradiction. Hence, $k := k + 1 = 1$ and $\mathsf{inv}(l_0) := \mathsf{inv}(\tau) = \{x \leq y\}$.

Once more, since $\mathsf{In} = \{l_1 \overset{y:=0}{\mapsto} l_0\} \neq \emptyset$, we go through the **while** loop of $\mathsf{In} \neq \emptyset \ \wedge\ (k = 0 \vee \mathsf{inv}(l_0) \neq \emptyset)$. Here, we bring the result of computations briefly. We choose $\tau := l_1 \overset{y:=0}{\mapsto} l_0$, we get $\mathsf{In} := \emptyset$, and since $\mathcal{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau = \emptyset \wedge \emptyset = \mathsf{true}$, $\mathsf{seen} := \{l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. $\mathsf{inv}(\tau) := \emptyset \wedge \emptyset \wedge \{y \leq x\} = \{y \leq x\}$ and $k := k + 1 = 2$, hence, $\mathsf{inv}(l_0) := \mathsf{max}(\{x \leq y\}, \{y \leq x\}) = \mathsf{max}(x \leq y, y \leq x) = \mathsf{true}$. At this point, since $\mathsf{In} := \emptyset$, we leave the **while** loop, and put $\mathcal{I}(l_0) := \mathcal{I}(l_0) \wedge \mathsf{simp}(\mathsf{inv}(l_0)) = \{y \leq 1\} \wedge \mathsf{simp}(\mathsf{true}) = \{y \leq 1\} \wedge \mathsf{true}$, which is equivalent to $y \leq 1$. So, we gain no new invariant for l_0.

Next, $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_0) = \{l_0 \overset{x:=0}{\mapsto} l_1,\ l_0 \overset{y>x}{\mapsto} l_1\}$. We choose $\tau := l_0 \overset{x:=0}{\mapsto} l_1$. $\mathsf{Out} := \mathsf{Out} \backslash \{\tau\} = \{l_0 \overset{y>x}{\mapsto} l_1\}$. $\tau \notin \mathsf{seen}$, and $\mathcal{I}(l_0) \wedge \mathsf{G}_\tau = y \leq 1$ which is not a contradiction. Hence, $\mathsf{seen} := \{l_0 \overset{x:=0}{\mapsto} l_1,\ l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. Then, we choose $\tau := l_0 \overset{y>x}{\mapsto} l_1$, and with the same reason, $\mathsf{seen} := \{l_0 \overset{y>x}{\mapsto} l_1,\ l_0 \overset{x:=0}{\mapsto} l_1,\ l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. $\mathsf{Out} := \mathsf{Out} \backslash \{\tau\} = \emptyset$, hence we leave this **while** loop, and put $i := i + 1 = 1$. $1 < 3$, so the **repeat** loop must be gone through once more.

The same process should be repeated again. The interesting part in this second round is that for $\tau := l_0 \overset{y>x}{\mapsto} l_1 \in {}^{\mathsf{in}}\mathsf{trans}(l_1, \mathcal{A})$, we get $\mathsf{inv}(\tau) := \{y > x\} \wedge \emptyset \wedge \emptyset = \{y > x\}$. For the other transition $\tau := l_0 \overset{x:=0}{\mapsto} l_1 \in {}^{\mathsf{in}}\mathsf{trans}(l_1, \mathcal{A})$, $\mathsf{inv}(\tau) := \{x \leq y\}$, and $\mathsf{inv}(l_1) := \mathsf{max}(\{x \leq y\}, \{y > x\}) = \{x \leq y\}$. In the end, $\mathcal{I}(l_1) := \emptyset \wedge \{x \leq y\} = x \leq y$. So, for this location we actually obtain a new invariant which is $x \leq y$. See Figure 2.

Then, $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_1) = \{l_1 \overset{y<x}{\mapsto} l_2,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. We choose $\tau := l_1 \overset{y<x}{\mapsto} l_2$, then $\mathsf{Out} := \{l_1 \overset{y:=0}{\mapsto} l_0\}$. $\tau \notin \mathsf{seen}$, and $\mathcal{I}(l_1) \wedge \mathsf{G}_\tau = x \leq y \wedge y < x$, which is a contradiction! This is shown in Figure 3(a). Therefore, the automaton is updated to $\mathcal{A} := \mathcal{A} \backslash \{l_1 \overset{y<x}{\mapsto} l_2\}$. We continue the algorithm with this new automaton. For this, see Figure 3(b). The other transition $l_1 \overset{y:=0}{\mapsto} l_0$ is already in seen. So this loop terminates here.

With this new automaton, for location l_2 we obtain: $\mathsf{In} := {}^{\mathsf{in}}\mathsf{trans}(l_2, \mathcal{A}) = \emptyset$. Hence, the **while** loop of $\mathsf{In} \neq \emptyset \ \wedge\ (k = 0 \vee \mathsf{inv}(l_2) \neq \emptyset)$ can not be entered. There is also no transition out of this location.

In the end we obtain $\mathcal{A} := \mathcal{A} \backslash \{l_1 \overset{y<x}{\mapsto} l_2\}$, and $\mathcal{I}_\mathcal{A} := \{l_0 \mapsto \{y \leq 1\},\ l_1 \mapsto \{x \leq y\},\ l_2 \mapsto \emptyset\}$. Figure 3(b).

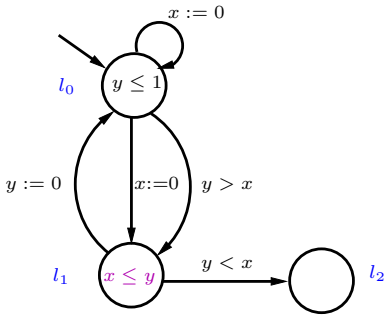

Fig. 2. New invariants.

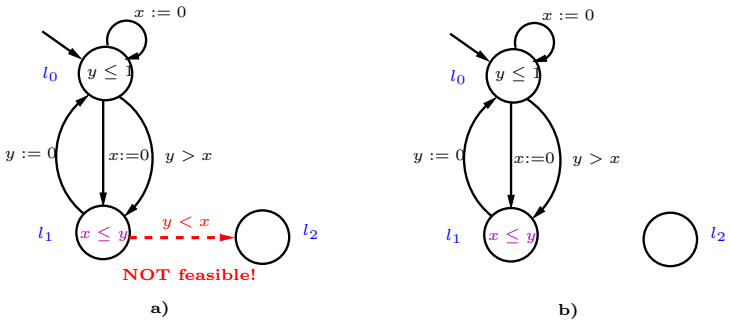

Fig. 3.

4 Predicate abstraction

After pruning the original timed automaton and creating new invariants at each control location, using the CIPM algorithm, we start abstracting the model. Since, the model is still too big to do reachability analysis on. In this section, we introduce a method for using the extracted invariants to build an abstraction of the model. Our abstraction will be an over-approximation of the original timed automaton.

4.1 The First Abstraction

We consider the abstract states to be pairs of control locations and conjunctions of (negated) predicates. Set of predicates consist of the clock constraints which are derived from the CIPM algorithm.

A *cube* q over $P = \{p_0, ..., p_n\}$, is a conjunction $\bigwedge_{0 \leq i \leq n} \tilde{p_i}$ over the elements of P and their negations, i.e. each $\tilde{p_i}$ is either equivalent to p_i or otherwise it is equivalent to its negation $\bar{p_i}$. For example $p_0 \wedge \bar{p_1} \wedge \bar{p_2} \wedge p_3$ is a cube over $\{p_0, p_1, p_2, p_3\}$. $\mathsf{cube}(P)$ denotes the set of all cubes over P. [3]

We use technique of *predicate abstraction* [8] to abstract timed models. Assume that $(\mathcal{A}, \mathcal{I}_\mathcal{A})$ is obtained from some timed automaton $\mathcal{A}_1$ using the CIPM algorithm (Algorithm 1). Based on this, we explain how to obtain a finite automaton, denoted $\mathsf{abst}_\mathcal{A}$, which is the *first abstraction* of $\mathcal{A}_1$ (and $\mathcal{A}$, by Theorem 1 and Note 2).

Without loss of generality, to avoid extra notations, we let $\mathcal{I}_\mathcal{A}(l_i)$ to represent $\mathsf{atom}(\mathcal{I}_\mathcal{A}(l_i))$.

[3] What we denote as *cube* here, describes the same thing as *minterm* in [16].

Definition 5 (States of $\mathsf{abst}_{\mathcal{A}}$). The set $\mathcal{I} := \bigcup_{0 \le i < \|A\|} \mathcal{I}_{\mathcal{A}}(l_i)$ is a collection of all invariants $\mathcal{I}_{\mathcal{A}}(l_i)$. Our predicate abstraction over $(\mathcal{A}, \mathcal{I}_{\mathcal{A}})$, denoted $\mathsf{abst}_{\mathcal{A}}$, is a finite state automaton where each state is of the form: $(l_i, \bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_i)} p \wedge \bigwedge_{p \in \mathcal{I} \setminus \mathcal{I}_{\mathcal{A}}(l_i)} \tilde{p})$, for some $0 \le i < \|\mathcal{A}\|$.

The intuition behind our abstraction is that since by Theorem 3, at each configuration $\langle l_i, u \rangle$, $u \models \mathcal{I}_{\mathcal{A}}(l_i)$, therefore the corresponding abstract sate should satisfy all predicates in $\mathcal{I}_{\mathcal{A}}(l_i)$, i.e. $\bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_i)} p$ should hold. Therefore, the abstraction would be more faithful if we pair up each location l_i with $\bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_i)} p$ as the base, and then conjunct this with cubes over the rest of the predicates, which are coming from $\mathcal{I} \setminus \mathcal{I}_{\mathcal{A}}(l_i)$.

We define cube_i as the set of all cubes over $\mathcal{I} \setminus \mathcal{I}_{\mathcal{A}}(l_i)$ which are satisfiable in conjunction with the predicates in $\mathcal{I}_{\mathcal{A}}(l_i)$, i.e.

$$\mathsf{cube}_i := \{q \mid q \in \mathsf{cube}(\mathcal{I} \setminus \mathcal{I}_{\mathcal{A}}(l_i)) \text{ and } \; q \wedge (\bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_i)} p) \text{ is satisfiable}\}$$

Note 3 We denote by (l_i, q) the abstract state $(l_i, (\bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_i)} p) \wedge q)$, when $q \in \mathsf{cube}_i$. (l_i, q) abstracts (approximates) all configurations $\langle l_i, u_i \rangle$ where $u_i \models q$.

Since there exist lots of well established tools for satisfiability checking of *difference logic* (DIFF), also for finding satisfiable cubes over a certain set of predicates in theories such as DIFF and EUF, we leave the satisfiability analysis to these tools. Examples of this are DDD [18], *AllSAT* [16] and DPLL(T) [21], which are all applicable here. Using these tools we can remove unsatisfiable cubes. Among these, *AllSAT* is a very good candidate for our goal. Given a set of predicates P and a subset of it $\mathcal{I}_{\mathcal{A}}(l_i)$, the inquiry then would be about finding the set of all cubes $c \in P \setminus \mathcal{I}_{\mathcal{A}}(l_i)$ such that $c \wedge \mathcal{I}_{\mathcal{A}}(l_i)$ is $T-$satisfiable, where T is the theory of difference logic. This way, we can find all the satisfiable cubes, automatically.

Example. We continue with the example of Section 3.2. To make it more expressive (for our approach), we assume that l_2 is originally given the invariant $y < x$, i.e. $\mathcal{I}(l_2) = \{y < x\}$. This will not deviate the final result of the CIPM except that we will have $\mathcal{I}_{\mathcal{A}}(l_2) = \{y < x\}$ (Figure 4).

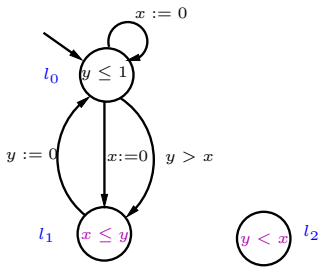

Fig. 4. Location l_2 with a new invariant.

Therefore, $\mathcal{I}_{\mathcal{A}}(l_0) = \{y \le 1\}$, $\mathcal{I}_{\mathcal{A}}(l_1) = \{x \le y\}$, $\mathcal{I}_{\mathcal{A}}(l_2) = \{y < x\}$ and hence, $\mathcal{I} = \bigcup_{0 \le i < \|A\|} \mathcal{I}_{\mathcal{A}}(l_i) = \{y \le 1, x \le y, y < x\}$. We represent the corresponding invariant of each

location l_i, with p_i, e.g. $p_0 := y \leq 1$. So, by definition:

$$\begin{aligned}
\mathsf{cube}(\mathcal{I}\backslash\mathcal{I}_{\mathcal{A}}(l_0)) &= \{p_1 \wedge p_2, \bar{p_1} \wedge p_2, p_1 \wedge \bar{p_2}, \bar{p_1} \wedge \bar{p_2}\} \\
\mathsf{cube}(\mathcal{I}\backslash\mathcal{I}_{\mathcal{A}}(l_1)) &= \{p_0 \wedge p_2, \bar{p_0} \wedge p_2, p_0 \wedge \bar{p_2}, \bar{p_0} \wedge \bar{p_2}\} \\
\mathsf{cube}(\mathcal{I}\backslash\mathcal{I}_{\mathcal{A}}(l_2)) &= \{p_0 \wedge p_1, \bar{p_0} \wedge p_1, p_0 \wedge \bar{p_1}, \bar{p_0} \wedge \bar{p_1}\}
\end{aligned}$$

Obviously, some of these combinations, for instance $p_1 \wedge p_2$, are unsatisfiable. After removing such combinations, and taking away the '$\wedge$' connectives for simplicity, we obtain: $\mathsf{cube}_0 = \{\bar{p_1}p_2, p_1\bar{p_2}\}$, $\mathsf{cube}_1 = \{p_0\bar{p_2}, \bar{p_0}\bar{p_2}\}$, and $\mathsf{cube}_2 = \{p_0\bar{p_1}, \bar{p_0}\bar{p_1}\}$. The corresponding states of our abstraction, abst_A, are illustrated in Figure 5. We will see later that the red dashed line, represents the unreachable states.

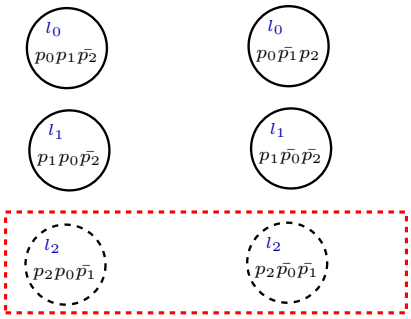

Fig. 5. The initial abstract states

Definition 6 (Transitions of $\mathsf{abst}_{\mathcal{A}}$). There is a transition, in $\mathsf{abst}_{\mathcal{A}}$, from a state (l_i, q) to a state (l_j, q'), for $q \in \mathsf{cube}_i$ and $q' \in \mathsf{cube}_j$, only if one of the conditions below, hold:

- there is a (non-idle) transition $\langle l_i, u_i\rangle \overset{\tau}{\mapsto} \langle l_j, u_j\rangle$ for two valuations u_i and u_j, where $u_i \models q$ and $u_j \models q'$, or
- if l_j is identical to l_i, and there is a delay transition $\langle l_i, u_i\rangle \overset{d}{\mapsto} \langle l_i, u_i + d\rangle$ for some valuation u_i such that $u_i \models q$ and $u_i + d \models q'$.

This with respect to Theorem 3 is equivalent to:

$$\begin{aligned}
\mathsf{next}(l_i, q) := \quad & \{(l_j, q') \mid \exists \tau \text{ or } d : \ \langle l_i, u_i\rangle \overset{\tau/d}{\mapsto} \langle l_j, u_j\rangle \text{ such that} \\
& u_i \models (\bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_i)} p) \wedge q \ \text{ and } \ u_j \models (\bigwedge_{p \in \mathcal{I}_{\mathcal{A}}(l_j)} p) \wedge q'\}.
\end{aligned}$$

We remind that τ is a discrete and d is a delay transition.

An automaton $\mathcal{A}_1$ is an *over-approximation* of $\mathcal{A}$ if $\mathcal{A}_1 \doteq \mathsf{abst}_{\mathcal{A}}$ and moreover for any transition between two (possibly identical) states of $\mathcal{A}$, there will be a transition in $\mathcal{A}_1$ between their corresponding abstract states, where the initial abstract state is the abstraction of the initial concrete state, and vice-versa.

Theorem 4. *If* $\mathsf{CIPM}(\mathcal{A}_1) = (\mathcal{A}, \mathcal{I}_{\mathcal{A}})$, *then* $\mathsf{abst}_{\mathcal{A}}$ *is an over-approximation of* $\mathcal{A}_1$.

Proof. – Assume there is a discrete transition τ from a control location l_i to a control location l_j in $\mathcal{A}_1$. Then, by definition, there exist valuations u_i and u_j such that $u_i \models \mathcal{I}(l_i)$

and $u_j \models \mathcal{I}(l_j)$. Therefore, $\langle l_i, u_i\rangle \overset{\tau}{\mapsto} \langle l_j, u_j\rangle$ in $\mathcal{S}_{\mathcal{A}1}$. According to Theorem 3, $u_i \models \mathcal{I}_{\mathcal{A}}(l_i)$ and $u_j \models \mathcal{I}_{\mathcal{A}}(l_j)$.

Now, for each predicate p in $\mathcal{I}\backslash\mathcal{I}_{\mathcal{A}}(l_i)$, either $u_i \models p$ or $u_i \models \bar{p}$. Conjunction of these (negated) predicates builds (only) one cube q in cube_i such that $u_i \models q$. Analogously, there is also a cube $q' \in \mathsf{cube}_j$ such that $u_j \models q'$. According to the definition, this would then mean that there is a transition from (l_i, q) to (l_j, q') in $\mathsf{abst}_{\mathcal{A}}$.

– Analogously, using Lemma 1, Theorem 2 and Note 3. ■

The red dashed line in Figure 5 depicts the unreachable states. Because, they correspond to some unreachable state in the concrete model, and wherefore these states are unreachable in the abstract model (cf. Theorem 4). After computing the corresponding transitions, we obtain Figure 6 as our *initial* abstract model.

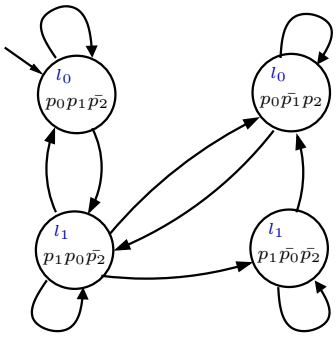

Fig. 6. $\mathsf{abst}_{\mathcal{A}}$, abstraction of $\mathcal{A}$.

In many existing techniques, control locations are not taken into abstraction. This in principle can lead to excessive number of spurious paths. In our approach we avoid many possible spurious paths in the abstract model by distinguishing locations and also sewing them with their *extended* invariants.[4] For example, in Figure 6, the cube $p_0p_1\bar{p_2}$ has appeared in two different sates, causing two different set of approximations, for l_0 and l_1.

Each timed automaton has a finite number of control locations, $\|\mathcal{A}\|$. We associate to each location l_i, at most $\|\mathsf{cube}_i\|$ abstract states. This way the number of abstract states, in the worst case, reaches $\Sigma_{0\leq i<\|\mathcal{A}\|}\|\mathsf{cube}_i\|$. In our last example, this number is $2+2+2=6$. This is when, as illustrated in Figure 6, the actual number of states reduces to 4, as a result of pruning the original model via CIPM. Without idle-transition detection and without pairing locations with invariants, one would have gotten $3 \times 4 = 12$ states, where 4 is the number of distinguished satisfiable cubes and 3 is the number of locations. Clearly, this number would rise to $3 \times 2^3 = 24$, when no satisfiability check on the cubes (through SAT-solvers) is done.

5 Conclusion

Our work provides an automatic approach for reducing the size of real timed automata by discovering and removing the idle transitions. We established conditions under which a transition is idle, i.e. it can never be traversed. Our algorithm also provides new invariants for control locations, based on the incoming and outgoing discrete transitions. In the second

[4] The idea of taking control locations into abstraction, is also used in [11]. However, there, neither predicates nor invariants are used to distinguish locations in the abstraction phase.

phase, we used the generated invariants to abstract the original model, using predicate abstraction technique. Here, control locations are inserted into the abstraction, through their invariants. This way, we provide a more honest abstraction. To illustrate the approach we used an example from [19]. It can easily be observed how our approach results in a more efficient and constructive abstraction.

Future work. Reachability analysis is our next step. See the Appendix for a summary on that. We have the idea of using interval arithmetic over intervals of real numbers $\mathbb{R}$, to discover possible transitions between two abstract states. How to refine the abstraction once a spurious path is found, is other direction for the extension of this work. It is our plan to implement this current approach in UPPAAL.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *TACAS'02*, pages 158–172, 2002.
3. Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. 2001.
4. Michael Colón and Tomás E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *CAV'98*, pages 293–304, 1998.
5. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. *POPL'77*, pages 238–252, 1977.
6. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design(FMCAD)*. Springer-Verlag, November 2002.
7. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *11th International Conference on Computer-Aided Verification (CAV'99)*. Springer-Verlag, 1999.
8. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
9. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL'04*, pages 232–244, 2004.
10. Timothy J. Hickey, Qun Ju, and Maarten H. van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, 2001.
11. Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in uppaal. In *Model Checking and Artificial Intelligence, MoChArt'06*, pages 51–66, 2006.
12. Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *CAV'05*, pages 39–51, 2005.
13. Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. *The Computing Research Repository, CoRR'06*, abs/cs/0612003, 2006.
14. Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.
15. Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A Symbolic Approach to Predicate Abstraction. In *CAV'03*, pages 141–153, 2003.
16. Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In *Proc. of Computer Aided Verification, CAV'06*, pages 424–437. Springer, 2006.
17. Kenneth L. McMillan and Nina Amla. Automatic Abstraction without Counterexamples. In *TACAS*, pages 2–17, 2003.
18. Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference Decision Diagrams. In *13th International Conference on Computer Science Logic (CSL)*, pages 111–125, 1999.
19. M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time System. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.
20. Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL*, pages 132–144, 2005.
21. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. Conference on Logics in Artificial Intelligence*, LNCS, pages 308–319, 2002.

Appendix

Using Interval Arithmetic for the Reachability Analysis

We propose to use *Interval Arithmetic* [10] for reachability analysis, in $\mathsf{abst}_{\mathcal{A}}$ The idea is to use conflict clauses as a tool to reason about (non-)reachability. Conflict clause analysis is exploited in [17] as a method for automatic abstraction and reachability analysis of unbounded models, without using counterexamples. Here, we follow the same purpose in real-time models. Let us explain the idea by an example:

Example 1. Let $p_1 := y \leq 1$ and $p_2 := y < x$ and $\phi := x = 0$, where p_1 and p_2 are predicates, $x, y \in \mathbb{R}+$ and ϕ is a formula. Let $\psi := p_1 \wedge p_2 \wedge \phi$. This formula is obviously unsatisfiable (we leave the satisfiability check, to existing tools, such as DDD in [18], DPLL(T) in [21], etc.). We now use interval analysis in order to find the conflict clause: from p_1, it is inferred that $y \in [0, 1]$, and from ϕ, $x \in [0, 0]$. From p_2, $y < x$, it is obtained that $y - x < 0$ and hence, $[0, 1] \ominus [0, 0] \subset (-\infty, 0)$, i.e. $[0, 1] \subset (-\infty, 0)$. Contradiction! As a result we can obtain: $\phi \wedge p_1 \Rightarrow \neg p_2$, similarly, $p_1 \wedge p_2 \Rightarrow \neg\phi$. Obviously, other combinations could be satisfiable too, e.g. $\neg p_1 \wedge p_2 \Rightarrow \neg\phi$.

We call this approach, *predicate exclusion*. To use this approach for reachability analysis of our abstract models, we will use the fact that there are only finite number of states, say, n, and hence to use the predicate exclusion technique, one will need to check n number of states, and for each state only the outgoing discrete transitions in its corresponding concrete state need to be tried. So, if there are at most k discrete transitions *in the whole* concrete model, [5] then the number of checks, in the worst case, will be $k \times n^2$. We will explain this in the sequel.

Note. Here, w.l.o.g, by reset constraints and clock constraints, we mean the atomic ones, unless it is explicitly states.

Interval Arithmetic. For any two real numbers a and b, the interval from a to b, denoted $[a, b]$, is the set of all real numbers between a and b. Formally, $[a, b] = \{r \in \mathbb{R} \mid a \leq r \leq b\}$. We replace "[" with "(" whenever the interval does not contain its border value, e.g. $(a, b] = \{r \in \mathbb{R} \mid a < r \leq b\}$, analogously is used ")" for "]". Interval operations $\oplus$ and $\ominus$ are respectively extensions of $+$ and $-$ over sets: $[a, b] \oplus [c, d] = \{r + s \mid r, s \in \mathbb{R} \text{ and } a \leq r \leq b,\ c \leq s \leq d\} = [a + c, b + d]$. Likewise, $[a, b] \ominus [c, d] = \{r - s \mid r, s \in \mathbb{R} \text{ and } a \leq r \leq b,\ c \leq s \leq d\} = [a - d, b - c]$. Details on this can be found in [10]. Intersection of intervals, $[a, b] \cap [c, d] = \{r \mid r \in \mathbb{R} \text{ and } a \leq r \leq b,\ c \leq r \leq d\}$. We extend this definition over clock and reset constraints, as:

Definition 7. Let us assume that our model contains only two clock variables x, y (of non-negative real value). We define the operation $\mathsf{interval}$ over clock and reset constraints, as:

- $\mathsf{interval}(x \leq y) := (x \in [0, y] \wedge y \in [0, \infty)) \ \vee \ (x \in [0, \infty) \wedge y \in [x, \infty))$
- $\mathsf{interval}(x < c) := x \in [0, c) \wedge y \in [0, \infty)$, for a constant value c
- $\mathsf{interval}(x := c) := x = [c, c] \wedge y \in [0, \infty)$, for a constant value c

[5] Actually, we only need to consider k to be the branching factor.

The operation, is defined similarly on other clock constrains, as well. Notice that whenever there is a clock variable, e.g. y in the last two items, which occurs in the concrete model but not in the constraints that $\mathsf{interval}$ is operating on, then we will let that variable to range over the interval of $[0, \infty)$.

Given two (or more) ranges of intervals, $\mathsf{interval}(t_1)$ and $\mathsf{interval}(t_2)$, we define the conjunction: $\mathsf{interval}(t_1) \sqcap \mathsf{interval}(t_2)$ as the conjunctions of all intervals I_x, where I_x is the intersection of the intervals of x in $\mathsf{interval}(t_1)$ and in $\mathsf{interval}(t_2)$. For example: if there are only two variables x and y in our concrete model, then $\mathsf{interval}(x \leq 1) \sqcap \mathsf{interval}(y < 2) = x \in [0, 1] \wedge y \in [0, 2)$ and $\mathsf{interval}(x \leq 1) \sqcap \mathsf{interval}(1 \leq x < 3) = x \in [1, 1] \wedge y \in [0, \infty)$. It can be observed that $\sqcap$ yields some intervals for each variable, which will satisfy both parties. This resulted interval is in a sence also the maximum segment (interval) which has this property.

Below, with the lemma we explain how reachability of a state can be entailed from some speculation on the intervals. The intuition behind it is that, when there is a transition between two states in abstract model, then there must be a concrete transition between the corresponding concrete states. This transition need to satisfy all the invariants of the source location and also to satisfy the clock constraint, g. Moreover, it should satisfy the invariants of the target location, since otherwise the transition can not be traversed. But whenever there is a reset for some clock value, then the new value of the clock should satisfy the invariant of the target location. Below, g/r denotes those constraints in g which do not have a common variable with reset values in r.

Lemma 6. Assume that $\mathsf{abst}_{\mathcal{A}}$ is an abstraction of $\mathcal{A}$ with respect to some set of predicates P. There is a transition from (l_i, q) to (l_j, q') in $\mathsf{abst}_{\mathcal{A}}$, i.e. $(l_j, q') \in \mathsf{next}(l_i, q)$, if and only if there exists a discrete/delay transition τ/d in $\mathcal{A}$ and some clock valuations u_i and u_j such that $\tau : \langle l_i, u_i \rangle \overset{\tau/d}{\mapsto} \langle l_j, u_j \rangle$, where

- $\mathsf{interval}(q) \sqcap \mathsf{interval}(g) \neq \emptyset$,
- $\mathsf{interval}(q) \sqcap \mathsf{interval}(g/r) \sqcap \mathsf{interval}(q') \neq \emptyset$ and
- $\mathsf{interval}(r) \sqcap \mathsf{interval}(q') \neq \emptyset$.

Proof. It can simply be observed that if all the conditions above hold then $(l_j, q') \in \mathsf{next}(l_i, q)$, according to its definition, because then there exist valuations u_i and u_j which satisfy the condition.

For the other direction, we have $u_i \models q$, and since τ can be traversed, $u_i \models g$. Hence, $\mathsf{interval}(q) \sqcap \mathsf{interval}(g) \neq \emptyset$. Analogously, we can derive $u_j \models q'$ and $u_j \models r$. As a sum up, $\mathsf{interval}(r) \sqcap \mathsf{interval}(q') \neq \emptyset$.

Once more, since the disceret transition τ can be traversed, the value of the variables which are not subject to reset (i.e. thoes appearing in g/r) have to satisfy not only g but also q'. Therefore, there must be some evaluation which satisfies q, g and q' for those variables which do not occur in r. As a result, $\mathsf{interval}(q) \sqcap \mathsf{interval}(g/r) \sqcap \mathsf{interval}(q') \neq \emptyset$. ■