Automated Invariant Generation for the Verification of Real-Time Systems

Bahareh Badban
Department of Computer and Information Science
University of Konstanz, Germany, Stefan Leue
Department of Computer and Information Science
University of Konstanz, Germany
and Jan-Georg Smaus*
Institut für Informatik
University of Freiburg, Germany

Abstract

We present an approach to automatically generating invariants for timed automata models. The CIPM algorithm that we propose first computes new invariants for timed automata control locations taking their originally defined invariants as well as the constrains on clock variables imposed by incoming state transitions into account. In doing so the CIPM algorithm also prunes idle transitions, which are transitions that can never be taken, from the automaton. We discsuss a prototype implementation of the CIPM algorithm as well as some initial experimental results.

1 Introduction

Predicate abstraction is an instance of the general theory of abstract interpretation [6]. It is a technique for generating finite abstract models of large or infinite state systems. This technique involves abstracting a concrete transition system using a set of formulas called *predicates*. Predicates usually denote some state properties of the concrete system. The predicate abstraction is conservative in the sense that if a property holds on the abstract system, there will be a concretization of the property that holds on the concrete system as well. Abstraction is defined by the value (true or false) of the predicates in any concrete state of the system [18]. This technique was first introduced by Graf and Saïdi [10] as a method for automatically determining invariant properties of infinite-state systems. As mentioned above, the idea of predicate abstraction is to generate a *finite state* abstraction of the system with respect to a finite set of predicates, which are mostly provided by the users themselves. Such an abstraction will have at most $2^{\|P\|}$ distinct states for a total number $\|P\|$ of predicates [16]. There are two obstacles to the practical use of the predicate abstraction approach:

1. The initial abstraction is *not fine enough*, and hence it is too abstract to be able to verify any property of the concrete model. In such a case the method relies on counter examples or proofs in the overall verification process to refine the abstraction [5], and as it is stated by Lahiri, et.al. in [15]:

    ”It is not clear if it is always preferable to compute the abstraction incrementally. But, we have observed that the refinement loop can often become the main bottleneck in these techniques (e.g. SLAM), and limits the scalability of the overall system.”

*The work of this author was partially funded by the German Research Council (DFG) through the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB TR14/AVACS).

2. The other obstacle is that the abstraction works with the predicates that are *provided by the user*. Hence, this method relies on the user's understanding of the system and on a trial-and-error process [15]. This phenomenon has been pointed out by Das and Dill in [7]:

> "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 [...]"

The purpose of our work is to provide support for an automated predicate abstraction technique for dense real-time models according to the timed automaton model of [1] by generating a more useful set of predicates than a manual, ad-hoc process would be able to provide. We analyze the behaviour of the system under verification to discover its local state invariants. During this analysis we remove idle transitions which are transitions that can never be traversed. We plan to incorporate the generated invariants into the abstraction phase of a counterexample guided abstraction refinement method for timed automata by using them as the *initial set of predicates* that is used to define an initial abstraction of the concrete model.

Related Work. How to discover predicates for use in the abstraction of real time system models has been widely discussed in the literature. Colón and Uribe [5] introduce an interactive method for predicate abstraction of real-time systems where a set of predicates called *basis* is provided by the user. As mentioned in the paper itself, this way the choice of abstraction basis is based on the user's understanding of the system. Therefore, generation of a suitable abstraction basis relies on trial-and-error. Möller et. al. in [21] introduce a method which is based on identifying a set of predicates that is fine enough to distinguish between any two clock regions and which creates a strongly preserving abstraction of the system. Refinement of the abstraction is accomplished using an analysis of the spuriousness of counter examples. Also in [7] Das and Dill use the spurious trace in discovering predicates for the predicate abstraction. Das et. al. in [8] introduce $Mor\phi^{--}$ which is a prototype for the verification of invariants in predicate abstraction. McMillan et. al. in [14, 19] and Henzinger, et. al. in [11] use interpolation to detect feasibility of the abstract trace and also to extract predicates from the proof for use in the abstraction. McMillan and Amla [20] introduce a proof-based automatic abstraction. The slicing approach of [13, 23] is a recent method with the same intention of reducing the state space of timed automata models. However, it is based on static analysis and not on a semantic interpretation of the automaton structure.

Another direction of research in the field of predicate abstraction addresses symbolic techniques. In this method a decision procedure takes a set of predicates P and symbolically executes a decision procedure on all the subsets over P. This results in a directed graph which represents the answer to a predicate abstraction query. The method aims at reducing the number of decision procedure calls since this number often tends to be extremely large [15]. In the first step, the first-order formula is encoded into some equi-satisfiable Boolean formula, and in the next step it is verified using a SAT solver. Examples of different symbolic methods are in [18, 15, 17, 16].

In [12] Hoffmann et. al. use the technique of predicate abstraction in order to obtain search heuristics to be used in directed model checking of safety properties. 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.

Structure of the Paper. The paper is organized as follows. Section 2 provides some preliminary definitions on real-time automata. The semantics of timed automata and their possible transitions are discussed in section 2.1. Section 3 introduces our method of creating new invariants. The respective algorithm, called CIPM, is explained in Section 3.1. In Section 3.2 we illustrate the algorithm by an example which is used by Möller et. al. in [21]. Section 4 describes an initial implementation of our approach. Finally, Section 5 presents concluding remarks and discusses future work.

2 Preliminary Definitions

In this section we reiterate the classical definition of timed automata according to [4, 1]. Additional concepts and notations which will be used throughout the paper are also introduced in this section.

A *timed automaton* consists of a finite state automaton together with a finite set of *clocks*. Clocks are non-negative real valued variables which keep track of the *time* elapsed since the last reset operation performed on the respective clock. The finite state automaton describes the system *control* states and its transitions. Initially, all clocks are set to 0. All clocks evolve at the same speed. A *configurations* of the system is given by the current control location of the automaton 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 $d \in \mathbb{R}^{+}$[1], is a valuation which assigns to each clock x the value $u(x) + d$, i.e., it increases the value of all clocks by d. $\mathsf{G}(X)$ denotes the set of *clock constraints* g for a set X of clock variables. Each g is of the form

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

where $x \in X$, and t, called *term*, is either a variable in X or a constant in $\mathbb{R}^+$. By $var(g)$ we denote the set of all clock variables appearing in g. A timed automaton is then formally defined as follows:

Definition 1. *A timed automaton $\mathcal{A}$ is a tuple $\langle L, l_0, \Sigma, X, \mathscr{I}, E\rangle$ 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,*
- *$\mathscr{I} : L \mapsto \mathsf{G}(X)$ assigns to each location in L some clock constraint in $\mathsf{G}(X)$,*
- *$E \subset L \times \Sigma \times 2^X \times \mathsf{G}(X) \times L$ represents* discrete *transitions.*

[1] $\mathbb{R}^+$ is the set of all non-negative reals, including 0.

The clock constraint associated with each location $l \in L$ is called its *invariant*, denoted $\mathscr{I}(l)$. We later refer to these invariants as the *original* invariants. It requires that time can pass in a control location only as long as its corresponding invariant remains true. In other words, $\mathscr{I}(l)$ must hold whenever the current state is l.

We call a constraint g *atomic* if there exist two terms s and t such that g is equivalent to $s \leq t,\ t \leq s$ or their negation[2]. With each clock constraint g, we associate a set of its atomic sub-formulas, $\mathsf{atom}(g)$, 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 are conjunctions of (one or more) formulas of the form $x := c$ where c is a constant in $\mathbb{R}^+$. We similarly define the application of atom over reset constraints, 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. We define a set of unbounded constraints in a set A of constraints as: $^{\mathsf{un}}(A) := \{a \in A \mid a \text{ is unbounded}\}$.

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. For a set A, $u \models A$ if $u \models a$ for each $a \in A$. Given two constraints g and g', g entails g', denoted $g \Rightarrow g'$, if for any valuation u, $u \models g'$ if $u \models g$. For instance, $x \geq 3 \Rightarrow x > 2$. Based on this, we define a function join[3] as:

$$\mathsf{join}(g, g') := \begin{cases} g' & \text{if } g \Rightarrow g' \\ g & \text{if } g' \Rightarrow g \\ \mathsf{true} & \text{if } \neg g \Rightarrow g' \text{ or equivalently, } \neg g' \Rightarrow g \\ g \vee g' & \text{otherwise} \end{cases} \tag{1}$$

Intuitively, given any two constraints g and g', $\mathsf{join}(g, g')$ is equivalent to the weaker one. We further extended this definition over sets:

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

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

In a timed automaton the values of all clocks evolve at the same speed. Therefore, 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 fact leads us to the next property:

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

[2]The negation of $t \leq s$ is $s < t$.
[3]This operation is called *strong join* in [22].

Lemma 1. *If $u \models g$ for a valuation u and some* unbounded *atomic constraint g, then $u+d \models g$ for any $d \in \mathbb{R}^+$. 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). By definition this property also extends to sets of unbounded atomic constraints. □

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

1. *$g \Rightarrow \mathsf{join}(g, g')$ and $g' \Rightarrow \mathsf{join}(g, g')$.*

2. *if $u \models g$ for some valuation u, then $u \models \mathsf{join}(g, g')$.*

3. *if $u \models A$ (or $u \models B$) for some valuation u, then $u \models \mathsf{join}(A, B)$.*

Proof. The proofs of these properties can be sketched as follows:

1. This is obvious according to the definition of join.

2. By the definition of join the statement 2 is equivalent to statement 1.

3. $u \models A$, hence by definition, $u \models a$ for all $a \in A$. Therefore, according to the second item, for all $a \in A$ and $b \in B$, $u \models \mathsf{join}(a, b)$. Hence, by definition, $u \models \mathsf{join}(A, B)$. This holds, analogously, for when $u \models B$.

□

2.1 Semantics

We associate a transition system $\mathcal{S}_{\mathcal{A}}$ with each timed automaton $\mathcal{A}$[4]. 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 $\mathscr{I}(l)$, i.e. $u \models \mathscr{I}(l)$. $\langle l_0, u \rangle$ 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:

- *Delay transitions* allow time $d \in \mathbb{R}^+$ to elapse. The value of all clocks is increased by d leading to the transition $\langle l, u \rangle \overset{d}{\mapsto} \langle l, u + d \rangle$. This transition can take place only when the invariant of location l is satisfied along the transition.

- *Discrete transition* enabling a transition (c.f. Definition 1)[5]. In this case all clocks, except those which are reset, remain unchanged. This results in 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 (c.f. Definition 1).

[4]We work with nondeterministic timed automata.
[5]A transitions is *enabled* if it can be traversed from the source control location.

An *execution* of a system is a possibly infinite sequence of configurations $\langle l, u\rangle$ where each pair of two consecutive configurations corresponds 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 do not need 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 called source of τ, denoted $\mathsf{sorc}(\tau)$, and target of τ, denoted $\mathsf{tar}(\tau)$, respectively. $\mathsf{G}_{\tau/\mathsf{R}_\tau}$ (respectively $\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}$) represents the set of all atomic constraints in G_τ (respectively $\mathscr{I}(\mathsf{sorc}(\tau))$) which do not have a variable occurring in R_τ. For instance, if $\tau := \langle l, u\rangle \overset{x\leq y\wedge z<x+1,\ z:=0}{\mapsto} \langle l', u'\rangle$, then $\mathsf{G}_\tau = \{x \leq y,\ z < x+1\}$, $\mathsf{R}_\tau = \{z := 0\}$ and $\mathsf{G}_{\tau/\mathsf{R}_\tau} = \{x \leq y\}$. In this example, z occurs in R_τ. For this transition if $\mathscr{I}(l) = \{y < 2, z > 4\}$ then $\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau} = \{y < 2\}$.

For each discrete transition τ we define:

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

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

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

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

Lemma 3. *For each 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 occurring. These are variables which belong to $\mathsf{G}_{\tau/\mathsf{R}_\tau}$. 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 reasoning applies to ${}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau})$. Since u' has some new values for the reset variables in r it holds that $u' \models \overline{r}$ right at the time of the occurrence of the transition (notice that no time has elapsed yet). Therefore, $u' \models \overline{\mathsf{atom}}(\mathsf{R}_\tau)$. Summing these up results in $u' \models \mathsf{inv}(\tau)$. $\square$

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, ${}^{\mathsf{out}}\mathsf{trans}(l, \mathcal{A})$ as:*

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

Notice that this definition relies exclusively on discrete transitions. Since only a finite number of such transitions exists these two sets are well-defined.

We now define a *reduction system*. This system simplifies the (disjunction of) clock constraints. The intuition is that any valuation function u satisfies the left hand-side of the reduction step (denoted by $\longrightarrow$) if and only if it also satisfies the right-hand side.

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 are applicable on it. We apply the *reduction rules* of Definition 3 on ϕ as long as they can be applied, which means that the result of the application is not identical to the constraint itself. When the reduction process terminates, we call the result $\mathsf{simp}(\phi)$. Obviously, $\mathsf{simp}(\phi)$ is simplified.

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, $u \models \mathsf{simp}(\phi)$ as well. □

In the next section we introduce an automatic approach to creating new invariants and to reducing the size of the model by pruning away those transitions which can never be traversed and which hence have no impact on a reachability analysis of the model.

3 Creating New Invariants

In this section we present the CIPM algorithm which strengthens the given original invariants in each control location by analysing the incoming discrete transitions to that specific control location.

3.1 The Algorithm

The input of Algorithm 1 is a timed automaton. Without loss of generality we assume that each location is assigned a separate index between 0 and $\|\mathcal{A}\| - 1$, e.g. l_1, where $\|\mathcal{A}\|$ is the number of control locations in $\mathcal{A}$.

Before explaining the algorithm we introduce the notion of idleness of a transition which expresses that a transition will never be enabled.

Definition 4. *A discrete transition $\tau : \langle l, u\rangle \mapsto \langle l', u'\rangle$ is called* idle *if it can never be enabled.*

Amongst other reasons, a transition can be idle 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 (i.e., $u' \neg \models \mathscr{I}(l')$). For instance, if τ is the discrete transition $\langle l, u\rangle \overset{x \leq y}{\mapsto} \langle l', u'\rangle$, where $x > y + 3$ is invariant in location l, i.e. $\mathscr{I}(l) = \{x > y + 3\}$, then this transition is idle since the constraint $x \leq y$ is never fulfilled as long as we are in l.

The CIPM algorithm first collects the set $\mathscr{I}(l_i)$ of all the original invariants in each location l_i. It then selects each location l_i and collects its incoming transitions in ${}^{\mathsf{in}}\mathsf{trans}(l_i, \mathcal{A})$. The idle transitions within that set are detected using Lemma 5 and are deleted from the model. For each non-idle τ in ${}^{\mathsf{in}}\mathsf{trans}(l_i, \mathcal{A})$ the algorithm next computes $\mathsf{inv}(\tau)$. It thereby extracts all constraints that are propagated to location l_i when executing transition τ. Applying join to all propagated constraints yields $\mathsf{inv}(l_i)$ which defines the full set of constraints that are imposed on l_i by all of the transitions in ${}^{\mathsf{in}}\mathsf{trans}(l_i, \mathcal{A})$. Since l_i may also have some original invariant, $\mathscr{I}(l_i)$ is the conjunction of the original invariant and all of the previously computed imposed constraints on l_i. This is expressed in the algorithm by $\mathcal{I}_{\mathcal{A}}(l_i) := \mathscr{I}(l_i) \wedge \mathsf{simp}(\mathsf{inv}(l_i))$. Computing $\mathcal{I}_{\mathcal{A}}(l_i)$ may render some of the outgoing transitions of l_i idle. Therefore, the algorithm next checks all outgoing transitions of l_i for idleness again using Lemma 5 (1st item). It then removes all transitions detected as being idle. The set seen stores the traversed transitions. It is used to ensure that all transitions are checked for being idle only once.

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 $B = \{b_1, ...b_m\}$ [6].

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

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

Proof.

- $\mathscr{I}(\mathsf{sorc}(\tau))$ 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 $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ holds. By assumption, this 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, $u_i \models \mathscr{I}(\mathsf{tar}(\tau))$ too. Therefore, $u_i \models \mathsf{inv}(\tau) \wedge \mathscr{I}(\mathsf{tar}(\tau))$. This contradicts the assumption. So, τ is never enabled.

$\square$

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* $\mathcal{A}\doteq\mathcal{A}_1$*, and*
- *for each control location l,* $\mathsf{inv}(l)$ *consists of only unbounded constraints.*

Proof.

- In timed automata, the number of control locations ($\|\mathcal{A}\|$) is a finite number, say n. Hence, 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 a finite number of 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 a smaller number of idle transitions.
- The above argument can be easily derived from the definition of $inv(\tau)$ and the definition of $\mathsf{inv}(l)$ in the algorithm.

$\square$

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

The new constraint $\mathcal{I}_\mathcal{A}(l)$ implies the original invariant $\mathscr{I}(l)$ 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.

[6] $A \wedge \emptyset$ and $\emptyset \wedge A$ are equivalent to A.

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
    $\mathscr{I}(l_j) :=$ 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 $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau$ is a contradiction then
            $\mathcal{A} := \mathcal{A} \backslash \{\tau\}$                    %% the idle transition
          else
            $\mathsf{seen} := \mathsf{seen} \cup \{\tau\}$
        $\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \ \cup\ {}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \ \cup\ \overline{\mathsf{atom}}(\mathsf{R}_\tau)$
        if $\mathsf{inv}(\tau) \wedge \mathscr{I}(l_i)$ 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
            $\mathsf{inv}(l_i) := \mathsf{join}(\mathsf{inv}(l_i), \mathsf{inv}(\tau))$
      $\mathscr{I}(l_i) := \mathscr{I}(l_i) \wedge \mathsf{simp}(\mathsf{inv}(l_i))$
      $\mathsf{Out} := {}^{\mathsf{out}}\mathsf{trans}(l_i)$
      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 $\mathscr{I}(l_i) \wedge \mathsf{G}_\tau$ is a contradiction then
            $\mathcal{A} := \mathcal{A} \backslash \{\tau\}$                    %% the idle transition
          else
            $\mathsf{seen} := \mathsf{seen} \cup \{\tau\}$
    $i := i + 1$
  until $i < n$
  $\mathcal{I}_{\mathcal{A}} := \{l_i \mapsto \mathscr{I}(l_i)\}$
  return $(\mathcal{A}, \mathcal{I}_{\mathcal{A}})$
```

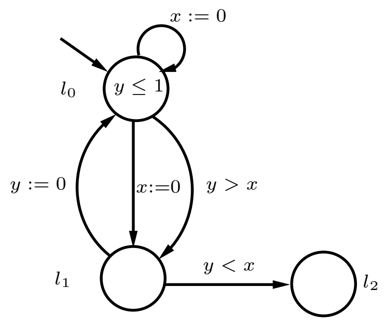

Figure 1: $\mathcal{A}$. In this timed automaton, x and y are clock variables.

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

Proof. $\mathcal{I}_{\mathcal{A}}(l) = \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$. According to the definition of $\Rightarrow$, we need to show that if $u \models \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$ then $u \models \mathscr{I}(l)$, for any valuation function u. Let $\mathscr{I}(l) = \{a_1, ...a_n\}$ 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 $u \models \mathscr{I}(l)$. □

The next theorem shows that $\mathcal{I}_{\mathcal{A}}$ associates with 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 reachable 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 $\mathcal{I}_{\mathcal{A}}(l) = \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$, we need to prove that $u \models \mathscr{I}(l) \wedge \mathsf{simp}(\mathsf{inv}(l))$ where $\mathscr{I}(l)$ is the original given invariant of location l (cf. Definition 1). To this end, we show that $u \models \mathscr{I}(l)$ and $u \models \mathsf{simp}(\mathsf{inv}(l))$. The 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 steps according to whether $\langle l, u\rangle$ is reached by a discrete transition τ or whether 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, $u \models \mathsf{join}(\mathsf{inv}(l), \mathsf{inv}(\tau))$ 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, a 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

We illustrate the CIPM algorithm using an example taken from [21], cf. the timed automaton in Figure 1. In this model, x and y are clock variables. l_0, l_1 and l_2 are control locations. Our intention is to prune this automaton and to find a new set of invariants for each control location according to the CIPM algorithm.

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. $\mathscr{I}(l_0) = \{y \leq 1\}$. In other words, we can stay in l_0 only as long as the value of y does not exceed 1. Once this value has passed 1 then one of the outgoing transitions must be taken out of this state (e.g. $l_0 \overset{x:=0}{\mapsto} l_1$). For the other locations we have: $\mathscr{I}(l_1) = \mathscr{I}(l_2) = \emptyset$.

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, $\mathscr{I}(l_0) = \{y \leq 1\}$ and $\mathscr{I}(l_1) = \mathscr{I}(l_2) = \emptyset$. 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 $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau = \{y \leq 1\} \wedge \emptyset$. By definition this 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: ${}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) = \emptyset$, ${}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) = {}^{\mathsf{un}}(\{\underline{y \leq 1}\}) = \emptyset$ and $\overline{\mathsf{atom}}(\mathsf{R}_\tau) = \{x \leq y\}$, we derive: $\mathsf{inv}(\tau) := {}^{\mathsf{un}}(\mathsf{G}_{\tau/\mathsf{R}_\tau}) \ \cup \ {}^{\mathsf{un}}(\mathscr{I}(\mathsf{sorc}(\tau))_{/\mathsf{R}_\tau}) \ \cup \ \overline{\mathsf{atom}}(\mathsf{R}_\tau) = \{x \leq y\}$. Then, $\mathsf{inv}(\tau) \wedge \mathscr{I}(l_0) = \{x \leq y\} \wedge \{y \leq 1\} = x \leq y \wedge y \leq 1$ 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 $\mathscr{I}(\mathsf{sorc}(\tau)) \wedge \mathsf{G}_\tau = \emptyset \wedge \emptyset = \mathsf{true}$, $\mathsf{seen} := \{l_0 \overset{x:=0}{\mapsto} l_0,\ l_1 \overset{y:=0}{\mapsto} l_0\}$. $\mathsf{inv}(\tau) := \emptyset \wedge \emptyset \wedge \{y \leq x\} = \{y \leq x\}$ and $k := k + 1 = 2$, hence, $\mathsf{inv}(l_0) := \mathsf{join}(\{x \leq y\}, \{y \leq x\}) = \mathsf{join}(x \leq y, y \leq x) = \mathsf{true}$. At this point, since $\mathsf{In} := \emptyset$, we leave the **while** loop, and put $\mathscr{I}(l_0) := \mathscr{I}(l_0) \wedge \mathsf{simp}(\mathsf{inv}(l_0)) = \{y \leq 1\} \wedge \mathsf{simp}(\mathsf{true}) = \{y \leq 1\} \wedge \mathsf{true}$, 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 $\mathscr{I}(l_0) \wedge \mathsf{G}_\tau = y \leq 1$ 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 for 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 $\mathsf{inv}(l_1) := \mathsf{join}(\{x \leq y\}, \{y > x\}) = \{x \leq y\}$. In the end, $\mathscr{I}(l_1) := \emptyset \wedge \{x \leq y\} = x \leq y$. This means that 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 $\mathscr{I}(l_1) \wedge \mathsf{G}_\tau = x \leq y \wedge y < x$, 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 $\mathcal{I}_\mathcal{A} := \{l_0 \mapsto \{y \leq 1\},\ l_1 \mapsto \{x \leq y\},\ l_2 \mapsto \emptyset\}$.

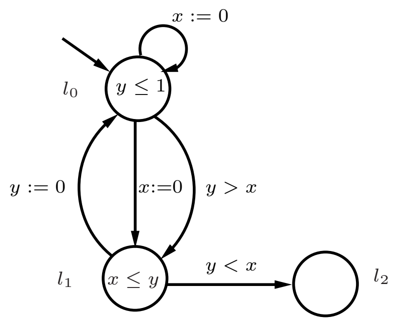

Figure 2: New invariants.

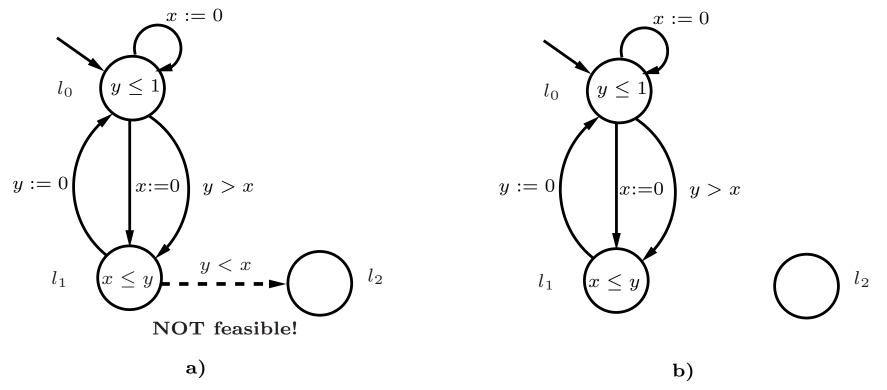

Figure 3:

Figure 3(b).

4 Implementation and Experimental results

We have developed a prototypical implementation of the CIPM algorithm in C++. The code takes a timed automaton in UPPAAL [3] syntax as input and computes the new invariants for each location, as well as removing the spurious transitions as shown in the pseudo-code of Alg. 1[7]. The actual implementation of CIPM thus consists of code for operating on the abovementioned datastructure, to remove transitions and modify invariants of the automaton.

The implementation consists of about 1000 lines of code. For checking whether some invariants togther constitute a contradiction (such a condition occurs in three places in the pseudo-code of Alg. 1), we use ICS [9], which is a solver for linear arithmetic. Likewise, we use ICS to check the implications occurring in the definition of join (see Equation 1).

Timed automata in UPPAAL are in some respects more general than the timed automata that we have defined in this paper. In particular, UPPAAL automata may have integer variables in addition to the real-valued clock variables. Integer variables only change their value when there is an explicit assignment. The presence of integer variables has repercussions on the

[7]For parsing the UPPAAL input file into a suitable C++ datastructure, we used code that was provided to the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB TR14/AVACS) gproject by Gerd Behrmann.

clocks since clock assignments, guards and invariants may involve linear expressions of those variables. E.g., there may be a guard $x \leq 2i + j$ where x is a clock variable and i and j are integer variables. In our current code, integer variables are ignored, but we plan to extend the implementation to cater for them.

Another feature of UPPAAL is that of parallel composition of automata (called *processes* in UPPAAL parlance) into a *system*. In the current implementation, we assume a system that is composed of just one process. Again, we plan to extend the implementation so that it works for systems with more than one process.

There is however one aspect where the automata we consider here are more general than UPPAAL automata: In UPPAAL, there are no disjunctive invariants. The invariant of a location, as well as the guard of a transition, is a sequence of equations and disequalities, interpreted as a conjunction. Therefore, we cannot directly implement the invariant $g \vee g'$ occurring in the definition of join, and we approximate it as the trivial invariant true. In some cases, we might be able to do better than that. E.g., $(x \geq 2 \wedge y \geq 1) \vee (x \geq 1 \wedge y \geq 2)$ implies $x \geq 1 \wedge y \geq 1$, which is stronger than true. However what the best approximation we can express with UPPAAL syntax is and how we can compute it is a nonobvious question left for future work as well. Another idea would be to split a location requiring a disjunctive invariant into two locations.

We have tested the implementation on the example of Fig. 1 and some other hand-designed examples. Our prototype tool transformed these examples in the expected way which increases our confidence that the proposed pseudo-code for CIPM accomplishes what we intend it to do.

5 Conclusion

Our work proposes the CIPM algorithm which accomplishes two goals. First, it automatically generates invariants for timed automata models. The algorithms computes new invariants in each control location of a timed automaton taking logical conditions on the original state invariants imposed by incoming transitions into account. Second, we defined the notion of idle transitions which helps in reducing the size of a timed automaton by eliminating transitions that can never be traversed.

We presented a preliminary implementation of the CIPM algorithms. At the current stage it is too early to talk about the performance of the implementation since the runtime for the examples including the one discussed above is, of course, negligible. However, the algorithm looks at each location and each transition at most once and thus its complexity should be low. How this would change if we ran the algorithm repeatedly on the same automaton is a different matter.

Future work includes the definition of a counterexample guided abstraction refinement technique using our proposed invariant generation approach extended by suitable predicate abstractions. Currently, we are incorporating the invariants computed by CIPM into an abstraction framework for timed automata. The idea is to couple each control location with its corresponding invariant and to use these invariants to determine a predicate abstraction for the respective pair of states. To illustrate the approach we considered an example from [21].

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*, 2002.

[3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems. Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, volume 3185 of *LNCS*, pages 200–236, Bertinoro, Italy, Sep 2004. Springer–Verlag.

[4] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer–Verlag, 2001.

[5] Michael Colón and Tomás E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *CAV'98*, pages 293–304, 1998.

[6] 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.

[7] 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.

[8] 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.

[9] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated canonizer and solver. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 246–249. Springer-Verlag, 2001.

[10] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[11] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL'04*, pages 232–244, 2004.

[12] 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.

[13] Agata Janowska and Pawel Janowski. Slicing of Timed Automata with Discrete Data. *Fundamenta Informaticae*, 72(1-3):181–195, 2006.

[14] Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *CAV'05*, pages 39–51, 2005.

[15] Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate Abstraction via Symbolic Decision Procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[16] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

[17] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A Symbolic Approach to Predicate Abstraction. In *CAV'03*, pages 141–153, 2003.

[18] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In *Proc. of Computer Aided Verification, CAV*, pages 424–437, 2006.

[19] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proc. of Computer Aided Verification, CAV*, pages 123–136, 2006.

[20] Kenneth L. McMillan and Nina Amla. Automatic Abstraction without Counterexamples. In *TACAS*, pages 2–17, 2003.

[21] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time System. *Electr. Notes Theor. Comput. Sci.*, 65(6), 2002.

[22] Sriram Sankaranarayanan, Michael Colón, Henny B. Sipma, and Zohar Manna. Efficient strongly

relational polyhedral analysis. In *VMCAI*, 2006.

[23] Uffe Sørensen and Claus Thrane. Slicing for Uppaal. Master's thesis, Aalborg university, 2008.