Automated Predicate Abstraction for Real-Time Models

Bahareh Badban[*] Stefan Leue
Department of Computer and Information Science, University of Konstanz, Germany
Bahareh.Badban@uni-konstanz.de Stefan.Leue@uni-konstanz.de

Jan-Georg Smaus
Institut für Informatik, Universität Freiburg, Germany
Smaus@informatik.uni-freiburg.de

Introduction Model checking has been widely successful in validating and debugging hardware designs and communication protocols. However, state-space explosion is an intrinsic problem which limits the applicability of model checking tools. To overcome this limitation software model checkers have suggested different approaches, among which abstraction methods have been highly esteemed. modern techniques. Among others, predicate abstraction is a prominent technique which has been widely used in modern model checking. This technique has been shown to enhance the effectiveness of the reachability computation technique in infinite-state systems. In this technique an infinite-state system is represented abstractly by a finite-state system, where states of the abstract model correspond to the truth valuations of a chosen set of atomic predicates. Predicate abstraction was first introduced in [8] as a method for automatically determining invariant properties of infinite-state systems. This technique involves abstracting a concrete transition system using a set of formulas called *predicates* which usually denote some state properties of the concrete system.

The practical applicability of predicate abstraction is impeded by two problems. First, predicates need to be provided manually [11, 7]. This means that the selection of appropriate abstraction predicates is based on a user-driven trial-and-error process. The high degree of user intervention also stands in the way of a seamless integration into practical software development processes. Second, very often the abstraction is too coarse in order to allow relevant system properties to be verified. This calls for abstraction refinement [6], often following a counterexample guided abstraction refinement scheme [5, 3]. Real time models are one example of systems with a large state space as time adds much complexity to the system. In this event, recently there have been increasing number of research to provide a means for the abstraction of such models. It is the objective of this paper to provide support for an automated predicate abstraction technique for concurrent dense real-time models according to the timed automaton model of [1]. We propose a method to generate an efficient set of predicates than a manual, ad-hoc process would be able to provide. We use the results from our recent work [2] to analyze the behavior of the system under verification to discover its local state invariants and to remove transitions that can never be traversed. We then describe a method to compute a predicate abstraction based on these state invariants. We use information regarding the control state labels as well as the newly computed invariants in the considered control states when determining the abstraction predicates. We have developed a prototype

[*]Corresponding author

Axel Legay and Azadeh Farzan (Eds.): International Workshop on Verification of Infinite-State Systems (INFINITY 2009)
EPTCS 10, 2009, pp. 36–43, doi:10.4204/EPTCS.10.3

tool that implements the invariant determination. Work is under way to also implement the computation of a predicate abstraction based on our proposed method. We plan to embed our approach into a comprehensive abstraction and refinement methodology for timed automata.

Related Work. An interactive method for predicate abstraction of real-time systems where a set of predicates called *basis* is provided by the user is presented in [6]. The manual choice of the abstraction basis depends on the user's understanding of the system. The work presented in [15, 16] proposes an abstraction 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. The basis predicates are discovered by spurious paths obtained through model-checking of the system. Also, in this approach the choice of the original set of predicates relies on the user's understanding of the system, as well as on the counterexample generation experiments. To the best of our knowledge, at the time of writing, there has been no research done on *automatically* generating invariants (predicates) for dense real time models, which will be the central contribution of our paper.

In the functional setting the CEGAR methodology based on the seminal paper [5] has been rather influential in the development of hard- and software verification methodologies, e.g., [3]. Abstraction predicate discovery based on the analysis of spurious counterexamples is at the heart of the work in [7]. The approaches presented in [10, 13] and in [9] use interpolation to detect feasibility of an abstract trace. [14] introduces a proof-based automatic predicate abstraction.

1 Preliminary Definitions and our Previous Results

Timed Automata. To have this article self-contained we need to briefly explain some of the results in [2]. A *timed automaton* [4, 1].consists of a finite state automaton together with a finite set of clock variables, simply called *clocks*, and a finite set of integer variables. In the notation we distinguish clock and integer variables only where necessary. Clocks are non-negative real valued variables which all increase at the same speed, while integers change only when there is an explicit assignment. Initially, all clocks are set to 0. A clock may be reset, but afterwards it immediately starts running again. The finite state automaton describes the system *control* states of the system, which are referred to as *locations*, as well as its transitions between locations. A *state* or configuration of the system has the form $\langle l,u \rangle$ where l is the current control location and u is a valuation function which assigns to each its current value. For $d \in \mathbb{R}^+$, we denote by $u+d$ a valuation that assigns to each clock x the value $u(x)+d$, i.e., it increases the value of all clocks by d, while the integer variables remain unchanged. $\mathsf{G}(X)$ denotes the set of (clock or integer) *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 linear integer expression, which is an expression of the form $c + \sum_{i=1}^{n} c_i \cdot x_i$ where the x_i are integer variables and c and c_i are integer constants.[1] We usually write $s < t$ for $\neg\, t \leq s$. By $var(g)$ we denote the set of all clock variables appearing in g. Formally, a timed automaton $\mathscr{A}$ is a tuple $\langle L, l_0, \Sigma, X, \mathscr{I}, E \rangle$ where

- L is a finite set of *(control) locations*. $l_0 \in L$ is the initial location.
- Σ is a finite set of labels, called *events* or *channels*.
- X is a finite set of variables.
- $\mathscr{I} : L \longrightarrow \mathsf{G}(X)$ assigns to each location in L some constraint in $\mathsf{G}(X)$.

[1] The restriction to integers does not constitute a loss of generality [1, Section 4.1].

- $E \subset L \times \Sigma \times 2^X \times \mathsf{G}(X) \times L$ represents *discrete* transitions.

The 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. Time can pass in a control location l only as long as $\mathscr{I}(l)$ remains $\mathtt{true}$, i.e. $\mathscr{I}(l)$ must hold whenever the current location is l. The semantics of a nondeterministic timed automaton $\mathscr{A}$ is defined by a *transition system* $\mathscr{S}_{\mathscr{A}}$. States or configurations of $\mathscr{S}_{\mathscr{A}}$ are pairs $\langle l,u\rangle$, where $l \in L$ is a control location of $\mathscr{A}$ 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 $\mathscr{S}_{\mathscr{A}}$ if l_0 is the initial location.
Transitions. For each transition system the system state changes by:

- *Delay transitions*, denoted by d, which allow time $d \in \mathbb{R}^+$ to elapse. The value of all clocks is increased by d leading to the transition $\langle l,u\rangle \xrightarrow{d} \langle l,u+d\rangle$. [2] This transition can take place only when the invariant of location l is satisfied along the transition, i.e. $\forall d' \leq d : u+d' \models \mathscr{I}(l)$.
- *Discrete transitions*, denoted by τ, which enable a transition. A transition τ is *enabled* when the current clock valuation satisfies G_τ. When τ is executed, all variables, except those which are reset, remain unchanged. This results in the transition $\tau := \langle l,u\rangle \xrightarrow{a,g,r} \langle l',u'\rangle$ where a is an event, g is a guard and r is a reset.

An *execution* of a system is a possibly infinite sequence of states $\langle l,u\rangle$ where each pair of two consecutive states corresponds to either a discrete or a delay transition.

Creating New Invariants by CIPM**.** Here, we explain briefly the CIPM algorithm from [2]. This algorithm strengthens the given original invariants in each control location by analysing the incoming discrete transitions to that specific control location; It also reduces the size of the model by pruning away those transitions which can never be traversed. The input of the CIPM algorithm is a timed automaton $\mathscr{A}$, the output is $\mathscr{A}$'s pruned version together with a set of new invariants for $\mathscr{A}$.

A discrete transition $\tau : \langle l,u\rangle \longrightarrow \langle l',u'\rangle$ is called *idle* if it can never be enabled. Amongst other reasons, a transition can be idle when the constraint over the transition is unsatisfiable, or when the valuation function obtained from the transition does not satisfy the invariant of the target location, which means that $u' \not\models \mathscr{I}(l')$. For instance, if τ is the discrete transition $\langle l,u\rangle \xrightarrow{x \leq y} \langle l',u'\rangle$ where $x > y+3$ is an invariant in location l, then this transition is idle since the constraint $x \leq y$ is never satisfied.

At each control location l_i, CIPM first collects the set $\mathscr{I}(l_i)$ of all the original invariants, and then accumulates all its incoming transitions in $^{\mathsf{in}}\mathtt{trans}(l_i,\mathscr{A})$. The idle transitions within these sets are identified and are deleted from the model.

For each non-idle τ in $^{\mathsf{in}}\mathtt{trans}(l_i,\mathscr{A})$ the algorithm next computes all propagated constraints into l_i. Since l_i may also have some original invariant, the new invariant, i.e. $\mathscr{I}_{\mathscr{A}}(l_i)$, is the conjunction of the original invariant and all of the previously computed imposed constraints on l_i. Computing $\mathscr{I}_{\mathscr{A}}(l_i)$ 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. It then removes all transitions detected as being idle. Two timed automata $\mathscr{A}$ and $\mathscr{A}_1$ are *equivalent*, denoted $\mathscr{A} \doteq \mathscr{A}_1$, if they differ only in some idle transitions.

Theorem 1.1 CIPM *always terminates. It also satisfies the following properties:*

- *if* $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A},\mathscr{I}_{\mathscr{A}})$ *then* $\mathscr{A} \doteq \mathscr{A}_1$.
- *If* $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A},\mathscr{I}_{\mathscr{A}})$*, then* $u \models \mathscr{I}_{\mathscr{A}}(l)$*, for each reachable state* $\langle l,u\rangle$ *in* $\mathscr{S}_{\mathscr{A}1}$*. In other words,* $\mathscr{I}_{\mathscr{A}}(l)$ *is invariant in* l*.*

[2] Recall that the integer variables remain unchanged.

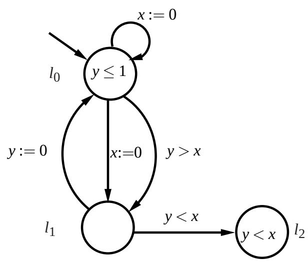

Figure 1: Example from [15]

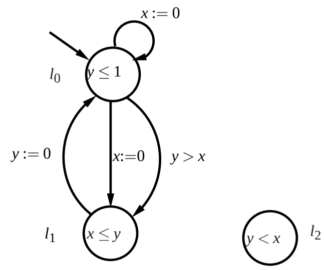

Figure 2: After applying CIPM

Networks of Timed Automata. CIPM can also be used to treat networks of timed automata in which several parallel automata synchronize with one another via synchronous message passing. Transitions associated with emitting or receiving a message of type a are labeled with $!a$ or $?a$, respectively. The intuitive semantics of a synchronous message passing is such that the message sending and the message receiving primitives are blocking and executed in a rendez-vous manner.

Formally, the semantics of this kind of synchronization is defined as follows. Let $A = \langle \overline{L}, \overline{l^0}, \Sigma, X, \mathscr{I}, E \rangle$ be a parallel composition of n timed automata $\mathscr{A}_1, \ldots, \mathscr{A}_n$, denoted by $\mathscr{A} = \mathscr{A}_1 \| \ldots \| \mathscr{A}_n$, where $\mathscr{A}_i := \langle L_i, l_i^0, \Sigma_i, X_i, \mathscr{I}_i, E_i \rangle$ for each $1 \leq i \leq n$ and for each two non-equal i and j $X_i \cap X_j = \emptyset$. For A we have $X = \bigcup_{1 \leq i \leq n} X_i$, $\Sigma = \bigcup_{1 \leq i \leq n} \Sigma_i$, and $\mathscr{I}(\overline{l}) = \bigwedge_{1 \leq i \leq n} \mathscr{I}(l_i)$ for $\overline{l} = (l_1, \ldots, l_n)$. The initial location is denoted by $\overline{l^0} = (l_1^0, \ldots, l_n^0)$. A state of the network is a configuration $\langle \overline{l}, u \rangle$ where $\langle l_i, u_i \rangle$ is a configuration in $\mathscr{A}_i$ and $u(x) = u_i(x)$ for each $x \in X_i$ and $1 \leq i \leq n$. $\overline{l}[l_i/l_i']$ denotes the replacement of l_i by l_i' in $\overline{l}$, which is $\overline{l}[l_i/l_i'] = (l_1, \ldots, l_{i-1}, l_i', l_{i+1}, \ldots, l_n)$. Delay transition in this systems is defined as before. Other transitions are:

- *Discrete transitions:* If $\langle l_i, u_i \rangle \xrightarrow{a,g,r} \langle l_i', u_i' \rangle$ then $\tau := \langle \overline{l}, u \rangle \xrightarrow{a,g,r} \langle \overline{l}[l_i/l_i'], u' \rangle$ is a discrete transition in the network model if $u'(x) = u_i'(x)$ for $x \in X_i$ and $u'(x) = u(x)$ for $x \notin X_i$.
- *Synchronization transitions:* If $\langle l_i, u_i \rangle \xrightarrow{!a,g,r} \langle l_i', u_i' \rangle$ and $\langle l_j, u_j \rangle \xrightarrow{?a,g,r} \langle l_j', u_j' \rangle$ then $\tau := \langle \overline{l}, u \rangle \longrightarrow \langle \overline{l}[l_i/l_i', l_j/l_j'], u' \rangle$ is a discrete transition in the network model if $u'(x) = u_k'(x)$ for $k \in \{i, j\}$ and $x \in X_k$, and $u'(x) = u(x)$ for $x \notin X_k$.

We first run the CIPM algorithm over each automaton individually. We then compose the pruned automata to obtain a pruned network. Conjuncting the newly generated invariants within the individual automata yields new invariants for the network:

Theorem 1.2 *Assume $\mathscr{A} = \mathscr{A}_1 \| \ldots \| \mathscr{A}_n$ is a network of timed automata where $\mathsf{CIPM}(\mathscr{A}_i) = (\mathscr{A}_i', \mathscr{I}_{\mathscr{A}'_i})$ for each $1 \leq i \leq n$, and $\mathscr{A}' = \mathscr{A}_1' \| \ldots \| \mathscr{A}_n'$. Then we will have $\mathscr{A} \doteq \mathscr{A}'$ and $\bigwedge_{1 \leq i \leq n} \mathscr{I}_{\mathscr{A}'_i}(l_i)$ is invariant in $\overline{l} = (l_1, \ldots, l_n)$.*

Example Figures 1 and 2 show an example of a timed automaton $\mathscr{A}$ in [15, 16], also the outcome of applying CIPM on it.

Example The example depicted in Figure 3 includes synchronization. Running the CIPM algorithm on $\mathscr{A}_1$ would result in the automaton $\mathscr{A}_2$ depicted in Figure 4. The algorithm would not change $\mathscr{B}_1$. However the parallel composition of $\mathscr{A}_2$ and $\mathscr{B}_1$ would lead to the parallel automata in Figure 4. This is because by Theorem 1.1 $\mathscr{A}_1 \| B_1 \doteq A_2 \| B_1$ and according to the definition of synchronization transitions $A_2 \| B_1 \doteq A_2 \| B_2$. As the figure depicts any configuration of the form $\langle (l_i, s_j), u \rangle$ for $i = 4$ or $j = 1$ is unreachable in $\mathscr{A}_2 \| \mathscr{B}_2$. Therefore, according to Theorem 1.2 any such configuration is also unreachable in $\mathscr{A}_1 \| \mathscr{B}_1$.

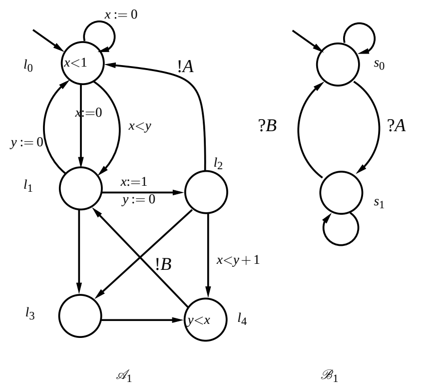

Figure 3: Parallel composition.

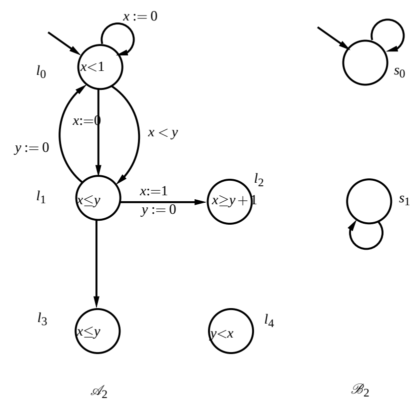

Figure 4: After applying CIPM.

2 Predicate Abstraction, New Results and the Ongoing Work

In this section, we introduce a method for using the invariants generated by CIPM in order to build an over-approximating *predicate abstraction* of the original timed automaton. We consider the abstract states not as Boolean vectors over the designated set of abstraction predicates, but rather as *pairs* of control locations and conjuncted, positive or negative predicates. In the sequel we will explain this in more detail.

A *cube* q over $P = \{p_0,...,p_n\}$, called a *minterm* in [12], 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 equivalent to either p_i or its negation $\overline{p_i}$. For example $x < 0 \wedge y > 2 \wedge z = 3$ is a cube over $\{x \geq 0, y \leq 2, z = 3\}$. $\mathsf{cube}(P)$ denotes the set of all cubes over P. In the sequel we assume that $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A}, \mathscr{I}_{\mathscr{A}})$ for a real time model $\mathscr{A}_1$, and our intention is to explain how to generate a predicate abstraction for $\mathscr{A}_1$. Without loss of generality, in the remainder of the paper we use $\mathscr{I}_{\mathscr{A}}(l_i)$ for $\mathsf{atom}(\mathscr{I}_{\mathscr{A}}(l_i))$.

States of $\mathtt{abst}_{\mathscr{A}}$**.** The set $\mathscr{I} := \bigcup_{0\leq i<\|A\|} \mathscr{I}_{\mathscr{A}}(l_i)$ is a collection of all invariants $\mathscr{I}_{\mathscr{A}}(l_i)$. Our predicate abstraction over $(\mathscr{A}, \mathscr{I}_{\mathscr{A}})$, denoted $\mathtt{abst}_{\mathscr{A}}$, is a finite state automaton where states are pairs like $(l_i, \bigwedge_{p\in\mathscr{I}_{\mathscr{A}}(l_i)} p \wedge \bigwedge_{p\in\mathscr{I}\setminus\mathscr{I}_{\mathscr{A}}(l_i)} \tilde{p})$ for $0 \leq i < \|\mathscr{A}\|$.

Spurious counterexamples when searching in the abstract state space are often due to invariant violations in the concrete model. In order to reduce the risk of generating spurious counterexamples we associate with each control location l_i its invariant as generated by CIPM. These invariants are gathered in $\mathscr{I}_{\mathscr{A}}(l_i)$. We first pair up each control location to its own invariant. Then we add the rest of the cubes from $\mathscr{I}\setminus\mathscr{I}_{\mathscr{A}}(l_i)$ to the pair. During construction of the abstraction each configuration $\langle l_i, u\rangle$ from the concrete model is abstracted to a abstract state in which $\mathscr{I}_{\mathscr{A}}(l_i)$ holds.

Let us consider cube_i as the set of all cubes over $\mathscr{I}\setminus\mathscr{I}_{\mathscr{A}}(l_i)$ which are satisfiable in conjunction with the predicates in $\mathscr{I}_{\mathscr{A}}(l_i)$:

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

For each $q \in \mathsf{cube}_i$ we denote by $[l_i, q]$ the abstract state $(l_i, (\bigwedge_{p\in\mathscr{I}_{\mathscr{A}}(l_i)} p) \wedge q)$. $[l_i, q]$ abstracts all configurations $\langle l_i, u_i\rangle$ in the concrete model $\mathscr{A}$ whose valuation u_i satisfies q, i.e. $u_i \models q$.

Example Let us continue with the first example (Figure 2). According to the example, we have $\mathscr{I}_{\mathscr{A}}(l_0) = \{y \leq 1\}$, $\mathscr{I}_{\mathscr{A}}(l_1) = \{x \leq y\}$, $\mathscr{I}_{\mathscr{A}}(l_2) = \{y < x\}$ and hence, $\mathscr{I} = \bigcup_{0\leq i<\|A\|} \mathscr{I}_{\mathscr{A}}(l_i) = \{y \leq 1, x \leq y, y < x\}$.

We use p_i to denote the invariant corresponding to the location l_i, therefore: $\mathsf{cube}(\mathscr{I}\setminus\mathscr{I}_{\mathscr{A}}(l_0)) = \{p_1 \wedge p_2, \overline{p_1} \wedge p_2, p_1 \wedge \overline{p_2}, \overline{p_1} \wedge \overline{p_2}\}$ $\mathsf{cube}(\mathscr{I}\setminus\mathscr{I}_{\mathscr{A}}(l_1)) = \{p_0 \wedge p_2, \overline{p_0} \wedge p_2, p_0 \wedge \overline{p_2}, \overline{p_0} \wedge \overline{p_2}\}$ $\mathsf{cube}(\mathscr{I}\setminus\mathscr{I}_{\mathscr{A}}(l_2)) = \{p_0 \wedge p_1, \overline{p_0} \wedge p_1, p_0 \wedge \overline{p_1}, \overline{p_0} \wedge \overline{p_1}\}$ Some of these combinations are unsatisfiable, for instance $p_1 \wedge p_2$. After removing such combinations and eliminating the '$\wedge$' symbol, for simplicity, we obtain: $\mathsf{cube}_0 = \{\overline{p_1}p_2, p_1\overline{p_2}\}$, $\mathsf{cube}_1 = \{p_0\overline{p_2}, \overline{p_0}\overline{p_2}\}$, and $\mathsf{cube}_2 = \{p_0\overline{p_1}, \overline{p_0}\overline{p_1}\}$. As illustrated in Figure 5 these three sets build an abstract model abst_A which consists of six states for example like $(l_0, p_0p_1\overline{p_2})$, $(l_1, p_1p_0\overline{p_2})$. As we shall see later on, the dashed line in this figure identifies unreachable states.

Transitions of $\mathsf{abst}_{\mathscr{A}}$ *In* $\mathsf{abst}_{\mathscr{A}}$ *we execute a transition from a state* $[l_i, q]$ *to a state* $[l_j, q']$ *only when one of the following conditions holds in the concrete model* $\mathscr{A}$*:*

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

Let $\mathsf{next}([l_i, q])$ denote the set of all successor states of $[l_i, q]$ in $\mathsf{abst}_{\mathscr{A}}$, then with respect to definition above:

$$\begin{aligned}\mathsf{next}([l_i,q]) :=& \{[l_j,q'] \mid \exists \tau \text{ or } d :\ \langle l_i,u_i\rangle \xrightarrow{\tau/d} \langle l_j,u_j\rangle \text{ such that} \\ & u_i \models (\bigwedge_{p\in\mathscr{I}_{\mathscr{A}}(l_i)} p) \wedge q \text{ and } u_j \models (\bigwedge_{p\in\mathscr{I}_{\mathscr{A}}(l_j)} p) \wedge q'\}. \end{aligned} \quad (2)$$

Recall that τ is a discrete and d is a delay transition.

Since $\mathsf{abst}_{\mathscr{A}}$ is an abstraction of $\mathscr{A}$, each of its transitions should have a counterpart in the original model $\mathscr{A}$. This means that whenever $[l_j, q'] \in \mathsf{next}([l_i, q])$, there must exist a non-idle transition from at least one of the corresponding concrete states of $[l_j, q]$ to that of $[l_j, q]')$. Such a transition needs to satisfy all the invariants of the source location and also all the invariants of the target location. Also if there is a reset for some variable, the new value of the respective variable should satisfy the invariant of the target location:

Lemma 2.1 *Assume that* $\mathsf{abst}_{\mathscr{A}}$ *is an abstraction of* $\mathscr{A}$ *with respect to some set of predicates* P*. There is a transition from* $[l_i, q]$ *to* $[l_j, q']$ *in* $\mathsf{abst}_{\mathscr{A}}$*, i.e.* $[l_j, q'] \in \mathsf{next}([l_i, q])$*, if and only if one of the conditions below holds:*

1. *there are two clock valuations u_i and u_j, and a non-idle transition $\tau : \langle l_i, u_i\rangle \longrightarrow \langle l_j, u_j\rangle$ in the concrete model such that:*
   (a) *$u_i \models q$ and $u_j \models q'$.*
   (b) *if $\mathsf{G}_\tau \neq \emptyset$ then $\mathsf{G}_\tau \wedge q$ is satisfiable,*
   (c) *if $\mathsf{G}_{\tau/\mathsf{R}_\tau} \neq \emptyset$ then $\mathsf{G}_{\tau/\mathsf{R}_\tau} \wedge q'$ is satisfiable,*
   (d) *if $\mathsf{R}_\tau \neq \emptyset$ then $\overline{\mathsf{atom}}(\mathsf{R}_\tau) \wedge q'$ is satisfiable,*
   (e) *for all variables $x \notin var(\mathsf{R}_\tau) \cup var(\mathsf{G}_\tau)$, $u_i(x) = u_j(x)$.*
2. *$l_i = l_j$ and $\exists d, u_i :\ \langle l_i, u_i\rangle \longrightarrow \langle l_i, u_i + d\rangle$ where $u_i \models q$ and $u_i + d \models q'$.*

The next theorem shows that in order to establish a predicate abstraction for the original concrete model $\mathscr{A}_1$ it is enough to do so for the pruned equivalent version obtained from an application of the CIPM algorithm:

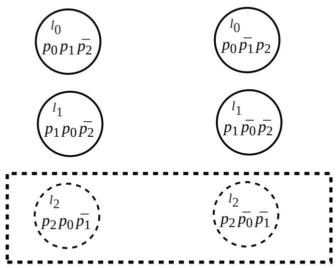

Figure 5: The states of $\mathsf{abst}_{\mathscr{A}}$

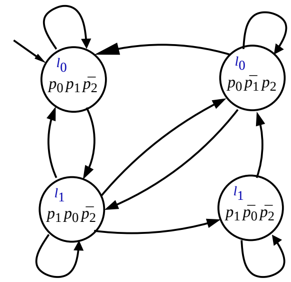

Figure 6: $\mathsf{abst}_{\mathscr{A}}$, predicate abstraction of $\mathscr{A}$.

Theorem 2.2 *If* $\mathsf{CIPM}(\mathscr{A}_1) = (\mathscr{A}, \mathscr{I}_{\mathscr{A}})$, *then* $\mathsf{abst}_{\mathscr{A}} \doteq \mathsf{abst}_{\mathscr{A}_1}$.

The cube $p_0 p_1 \overline{p_2}$ has caused two different abstract states in Figure 5. This is because p_0 and p_1 are invariants of l_0 and l_1, respectly, and therefore coupled with them in the abstract model. The dashed line in this figure depicts the set of unreachable abstract states of the first example. These states are unreachable since they correspond to some unreachable concrete states in $\mathscr{A}$ (cf. Lemma 2.1). Using Lemma 2.1 to compute the transitions in the abstract model, one would obtain Figure 6 as the initial predicate abstraction of $\mathscr{A}$. For instance from $(l_0, p_0 \overline{p_1} p_2)$ there is a transition to $(l_0, p_0 p_1 \overline{p_2})$ because the transition $\langle l_0, u \rangle \xrightarrow{x:=0} \langle l_0, u' \rangle$ fullfils Lemma 2.1.

In the following we give a simple succinctness analysis of our approach: Each timed automaton has a finite number of control locations, $\|\mathscr{A}\|$. We associate with each location l_i at most $\|\mathsf{cube}_i\|$ abstract states. This way the number of the abstract states is at most $\Sigma_{0 \leq i < \|\mathscr{A}\|} \|\mathsf{cube}_i\|$ in the worst case. In the example depicted in Figure 5, this number is $2+2+2=6$. By pruning the original model using CIPM and also with respect to Lemma 2.1 this number reduces to 4 abstract states, see Figure 6. With neither detecting the idle transitions nor pairing the control locations with their invariants, in the abstraction facet, one would have gotten $3 \times 4 = 12$ abstract states where 4 is the number of distinguished satisfiable cubes and 3 is the number of control locations. This number would have even raised to $3 \times 2^3 = 24$ abstract states if no satisfiability check on the cubes was done.

References

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126, 1994.

[2] B. Badban, S. Leue, and J.G. Smaus. Automated Invariant Generation for the Verification of Real-Time Systems. WING, 2009.

[3] T. Ball, A. Podelski, and S.K. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Proc. TACAS*, 2002.

[4] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. 2001.

[5] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, 2000.

[6] M. Colón and T.E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *Proc. CAV*, 1998.

[7] S. Das and D.L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Proc. FMCAD*, 2002.

[8] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.

[9] T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *Pro. POPL*, 2004.

[10] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *Proc. CAV*, 2005.

[11] S.K. Lahiri, T. Ball, and B. Cook. Predicate Abstraction via Symbolic Decision Procedures. *Logical Methods in Computer Science*, 3(2), 2007.

[12] S.K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *Proc. CAV*, 2006.

[13] K.L. McMillan. Lazy Abstraction with Interpolants. In *CAV*, 2006.

[14] K.L. McMillan and N. Amla. Automatic Abstraction without Counterexamples. In *TACAS'03*.

[15] M.O. Möller, H. Rueß, and M. Sorea. Predicate Abstraction for Dense Real-Time System. *ENTCS*, 2002.

[16] Maria Sorea. Lazy Approximation for Dense Real-Time Systems. In *Proc. of FORMATS-FTRTFT*, LNCS, 2004.