IEICE TRANS. INF. & SYST., VOL.E93–D, NO.5 MAY 2010

PAPER Special Section on Formal Approach

An Abstraction Refinement Technique for Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop

Takeshi NAGAOKA[†a)], *Nonmember*, **Kozo OKANO**[†], *and* **Shinji KUSUMOTO**[†], *Members*

SUMMARY Model checking techniques are useful for design of high-reliable information systems. The well-known problem of state explosion, however, might occur in model checking of large systems. Such explosion severely limits the scalability of model checking. In order to avoid it, several abstraction techniques have been proposed. Some of them are based on CounterExample-Guided Abstraction Refinement (CEGAR) loop technique proposed by E. Clarke *et al.*. This paper proposes a concrete abstraction technique for timed automata used in model checking of real time systems. Our technique is based on CEGAR, in which we use a counter example as a guide to refine the abstract model. Although, in general, the refinement operation is applied to abstract models, our method modifies the original timed automaton. Next, we generate refined abstract models from the modified automaton. This paper describes formal descriptions of the algorithm and the correctness proof of the algorithm.

key words: *model checking, timed automaton, model abstraction, CEGAR*

1. Introduction

A model checker checks if a given system modeled in a finite automaton satisfies given specifications by searching the finite transition system exhaustively. It sometimes has, however, limitation in scalability. In order to improve the scalability, a model abstraction technique is important [1]–[3].

In verification of real time systems, a timed automaton has widely been used [7], [8], which can describe behavior of realtime systems. In a timed automaton, real-valued clock constraints are assigned to its control state (called a location). Therefore, it has an infinite state space represented in a product of discrete state space made by locations and continuous state space made by clock variables. In the traditional model checking for a timed automaton, using the property that we can treat the state space of clock variables as a finite set of regions; we can perform model checking on timed automata models. However, the size of such regions increases exponentially with the number of clock variables; thus an abstraction technique is also needed.

Clarke *et al.* proposed an abstraction algorithm called CEGAR (CounterExample-Guided Abstraction Refinement) [1] shown in Fig. 1. The algorithm is used for abstraction of finite models [1], [2], hybrid systems [3], timed automata [11]–[13], and other models. In the CEGAR algorithm, we use a counter example produced by a model checker as a guide to refine excessively abstracted models. A general CEGAR algorithm consists of several steps. First, it abstracts the original model (the obtained model is called abstract model) and performs model checking on the abstract model. Next, if a counter example (CE) is found, it checks the counter example on the concrete model. If the CE is spurious, it refines the abstract model. The last step is repeated until the valid output is obtained. In the CEGAR loop, an abstract model must satisfy the following property; if the abstract model satisfies a given specification, the concrete model also satisfies it.

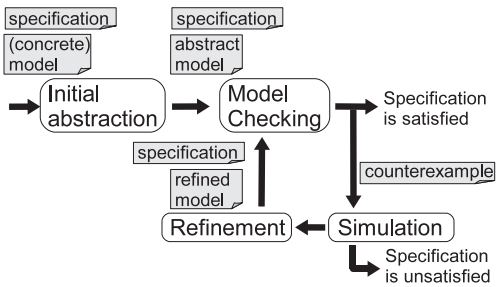


Fig. 1 General CEGAR algorithm.

This paper proposes a new concrete CEGAR algorithm for a timed automaton. The first step of the algorithm is abstraction, in which we delete all of time attributes from the given timed automaton. The obtained automaton is just a finite automaton preserving the transition relations of the timed automaton; therefore the obtained finite automaton is, in general, over-approximated of the original one. We restrict the class of the verification properties into reachability; thus if an abstract model satisfies a given property then the concrete model also satisfies the property.

In general, CEGAR algorithms [1]–[3], [11]–[13] directly transforms an abstract model using counter examples in the refinement step. Our proposed method, however, doesn't directly transform an abstract model. It first transforms the original model using counter examples and then it creates a new abstract model from it by removing clock attributes; thus our algorithm indirectly refines the abstract model. The algorithm transforms the original timed automaton by adding extra transitions and removing some transitions but it preserves the behavioral equivalence of the timed automaton and prevents the spurious counter examples. More concretely, it duplicates locations and transitions so that its abstract model can tell behavioral difference caused by clock values which affects the counter examples.

Manuscript received July 17, 2009.
Manuscript revised November 10, 2009.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan.
a) E-mail: t-nagaok@ist.osaka-u.ac.jp

DOI: 10.1587/transinf.E93.D.994

Consequently the obtained new abstract model does not accept the spurious counter example.

Related works [11]–[13] have proposed CEGAR based abstraction techniques for timed automata. Although these techniques mainly refine the abstract models by adding clock variables which have removed by abstraction, our refinement method modifies the original timed automata and produces the refined abstract model from the modified models, instead of adding clock variables.

The rest of the paper is organized as follows. In Sect. 2, some definitions are described. Section 3 gives our CEGAR algorithm and its application to a simple example. Section 4 proves the correctness of the algorithm. Section 5 concludes the paper.

2. Preliminaries

In this section, we give definitions of a timed automaton, semantics of the timed automaton, and the general CEGAR algorithm.

2.1 Timed Automaton

Definition 2.1 (Differential Inequalities on C)**.** *Syntax and semantics of a differential inequality E on a finite set C of clocks is given as follows:*
$E ::= x - y \sim a \mid x \sim a$,
where $x, y \in C$, a is a literal of a real number constant, and $\sim \in \{\leq, \geq, <, >\}$.
Semantics of a differential inequality is the same as the ordinal inequality.

Definition 2.2 (Clock Constraints on C)**.** *Clock constraints $c(C)$ on a finite set C of clocks are defined as follows:*
A differential inequality in on C is an element of $c(C)$.
Let in_1 and in_2 be elements of $c(C)$, $in_1 \wedge in_2$ is also an element of $c(C)$.

Definition 2.3 (Timed Automaton)**.** *A timed automaton $\mathscr{A}$ is a 6-tuple (A, L, l_0, C, I, T), where*
A: a finite set of actions;
L: a finite set of locations;
$l_0 \in L$: an initial location;
C: a finite set of clocks;
$I \subset (L \to c(C))$: a mapping from locations to clock constraints, called a location invariant; and
$T \subset L \times A \times c(C) \times \mathscr{R} \times L$,
where $c(C)$ is a clock constraint, called guards;
$\mathscr{R} = 2^C$: a set of clocks to reset.

A transition $t = (l_1, a, g, r, l_2) \in T$ is denoted by $l_1 \xrightarrow{a,g,r} l_2$. A map $\nu : C \to \mathbb{R}_{\geq 0}$ is called a clock assignment. We define $(\nu + d)(x) = \nu(x) + d$ for $d \in \mathbb{R}_{\geq 0}$. $r(\nu) = \nu[x \mapsto 0], x \in r$, where $\nu[x \mapsto 0]$ means the valuation that maps x into zero, is also defined for $r \in 2^C$. By N, a set of whole ν is denoted.

Definition 2.4 (Semantics of Timed Automaton)**.** *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, let $S = L \times N$ be a set of whole states of $\mathscr{A}$. The initial state of $\mathscr{A}$ shall be given as $(l_0, 0^C) \in S$.*
For a transition $l_1 \xrightarrow{a,g,r} l_2 \ (\in T)$, the following two kinds of transitions are derived from T. The former one is called an action transition, while the latter one is called a delay transition.

$$\frac{l_1 \xrightarrow{a,g,r} l_2, g(\nu), I(l_2)(r(\nu))}{(l_1, \nu) \overset{a}{\Rightarrow} (l_2, r(\nu))}, \qquad \frac{\forall d' \leq d \quad I(l_1)(\nu + d')}{(l_1, \nu) \overset{d}{\Rightarrow} (l_1, \nu + d)}$$

Definition 2.5 (A Semantic Model of Timed Automaton)**.** *For timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, an infinite transition system is defined according to the semantics of $\mathscr{A}$, where the model begins with the initial state. By $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$, the semantic model of $\mathscr{A}$ is denoted.*

In this paper, a state on a location l means an arbitrary semantic state (l, ν) such that ν satisfies l's invariant.

2.2 Zone Graph

In [7], a state space of timed automata, which has infinite semantic states, is represented as a finite state transition system called a zone graph. A zone $D \in c(C)$ is described as a product of finite differential inequalities on clock set C, which represents a set of clock assignments that satisfy all the inequalities. In this paper, we treat a zone D as a set of clock assignments $\nu \in \mathbb{R}^C_{\geq 0}$ (For a zone D, $\nu \in D$ means the assignment ν satisfies all the inequalities in D). In addition to this, using a location l and a zone D, we describe a set of semantic states as $(l, D) = \{(l, \nu) \mid \nu \in D\}$. Also, for an initial location l_0, a set of initial states is denoted by $(l_0, D_0) = \{(l_0, 0^C + d) | (l_0, 0^C) \overset{d}{\Rightarrow} (l_0, 0^C + d) \in \Rightarrow\}$.

Paper [7] also gives operation functions on zones, such as *up*, *and* and other functions, which represent elapsing time, intersection of time spaces and so on, respectively. For a given zone D, there is a minimal set of differential inequalities which is enough to represent D [7]. We use $Ineqset(D)$ to denote such a minimal set for D. $Ineqset(D)$ can be obtained by reduction operations on zones. A set of every state which satisfies an invariant $I(l)$ of location l is denoted by $(l, D_{I(l)})$ $(= \{(l, \nu) | I(l)(\nu)\})$.

When we create a zone graph from a timed automaton, we perform zone normalization called k-normalization [7], where $k : C \to \mathbb{N}$ is a clock ceiling, to prevent zones from increasing infinitely. The clock ceiling k is given by the maximal clock constants appearing in the automaton. In k-normalization, we represent zones that may contain arbitrarily large constants as a single representative zone. The details are given as follows; we remove the constraints of the form $x < m$, $x \leq m$, $x - y < m$, $x - y \leq m$ from the given zone, and also we replace the constraints of the form $x > m$, $x \geq m$, $x - y > m$, $x - y \geq m$ with $x > k(x)$, $x \geq k(x)$, $x - y > k(x)$, $x - y \geq k(x)$, where $x, y \in C$ and $m > k(x)$, respectively.

2.3 General CEGAR Algorithm

Model abstraction sometimes over-approximates an original model. It may produce spurious counter examples which are not actually counter examples in the original model. Paper [1] gives an algorithm called CEGAR (Counterexample-Guided Abstraction Refinement) (Fig. 1).

In the algorithm, at the first step (called Initial Abstraction), it over-approximates the original model. Next, it performs model checking on the abstract model. In this step, if the model checker reports that the model satisfies a given specification, we can conclude that the original model also satisfies the specification, because the abstract model is an over-approximation of the original model. If the model checker reports that the model does not satisfy the specification, however, we have to check whether the counter example detected is spurious counter example or not in the next step (called Simulation). In the Simulation step, if we find the counter example is valid, we stop the loop. Otherwise, we have to refine the abstract model to eliminate the spurious counter example, and repeat these steps until valid output is obtained.

3. Our CEGAR Algorithm for a Timed Automaton

Our proposed algorithm generates an abstract model $\hat{M}$ from a given timed automaton $\mathscr{A}$, and performs model checking on $\hat{M}$. $\hat{M}$ is in fact a finite automaton. If a counter example $\hat{\rho}$ (represented as a sequence of states and labels on $\hat{M}$) is detected by model checking, we check whether $\hat{\rho}$ is feasible on the concrete model $\mathscr{T}(\mathscr{A})$ or not at Simulation step (In this paper, for an abstract model $\hat{M}$ obtained from a timed automaton $\mathscr{A}$, we call the semantic model $\mathscr{T}(\mathscr{A})$ a concrete model of $\hat{M}$) . In this step, we obtain a set Π of sequences of transitions on $\mathscr{A}$ corresponding to $\hat{\rho}$, and check whether each path in Π is feasible on $\mathscr{T}(\mathscr{A})$ or not. If every path in Π is infeasible, the next step shall refine the model so that the counter example $\hat{\rho}$ becomes infeasible. Our algorithm does not directly refine $\hat{M}$ but it modifies $\mathscr{A}$ and then obtains a new abstract model from the modified timed automaton $\mathscr{A}$. Figure 2 shows flow of our CEGAR algorithm.

The proposed algorithm checks a property $\mathtt{AG} \bigwedge_{e \in E} \neg e$, where E ($\subset L$) of a timed automaton $\mathscr{A}$ is a set of error locations of the target system. The property means there is no path to locations in E from the initial state. Please note that any counter example of such a property can be represented in a finite length of sequence without infinite loops. Therefore, hereafter, we assume that counter examples are finite sequences.

3.1 Abstract Model

The proposed method abstracts a given timed automata $\mathscr{A} = (A, L, l_0, C, I, T)$ by removing clock variables from $\mathscr{A}$. Therefore, the obtained abstracted model $\hat{M}$ will be $(\hat{S}, \hat{s}_0, \hat{\Rightarrow})$, where $\hat{S} = L$, $\hat{s}_0 = l_0$.

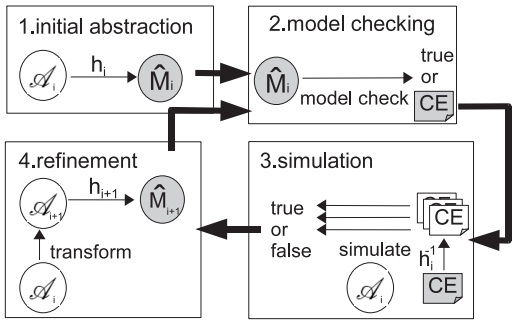


Fig. 2 Our proposed algorithm.

Here, we define the abstraction function $h : S \to \hat{S}$ which is a mapping from S to $\hat{S}$.

Definition 3.1 (Abstraction Function h)**.** *For a timed automaton $\mathscr{A}$ and its semantic model $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$, an abstraction function $h : S \to \hat{S}$ is defined as follows:*

$$h((l, v)) = l.$$

The inverse function $h^{-1} : \hat{S} \to 2^S$ of h is also defined as $h^{-1}(\hat{s}) = (l, D_{I(l)})$ where $\hat{s} = l$.

The abstraction function should be defined for each iteration of the refinement, because both the concrete model and abstract models are deformed. Let $\mathscr{A}_i$ and $\hat{M}_i$ be a timed automaton and an abstract model of i-th iteration, respectively. The abstraction function h_i for the i-th loop is defined in the similar way as Definition 3.1.

Symbols decorated with '^' represent those of an abstract model (i.e. $\hat{S}$ represents a state set of an abstract model). Definition 3.2 gives an abstract model $\hat{M}$ of a given timed automaton $\mathscr{A}$ using the abstraction function h defined in Definition 3.1.

Definition 3.2 (Abstract Model)**.** *An abstract model $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Rightarrow})$ of a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ and its semantic model $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ is defined as follows:*

- $\hat{S} = L$,
- $\hat{s}_0 = h(s_0)$, *and*
- $\hat{\Rightarrow} = \{(h(s_1), a, h(s_2)) \mid s_1 \overset{a}{\Rightarrow} s_2)\}$.

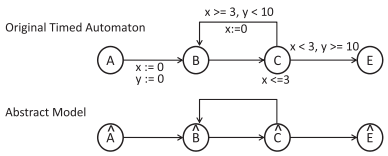
The i-th iteration of the refinement loop generates the i-th abstract model $\hat{M}_i = (\hat{S}_i, \hat{s}_{i,0}, \hat{\Rightarrow}_i)$ from the i-th timed automaton $\mathscr{A}_i = (A_i, L_i, l_{i,0}, C_i, I_i, T_i)$ by Definition 3.2.

Definition 3.3 (Abstract Counter Example)**.** *A counter example on $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Rightarrow})$ is a sequence of states of $\hat{S}$ and labels. An abstract counter example $\hat{\rho}$ of length n is represented in $\hat{\rho} = \langle \hat{s}_0 \overset{a_1}{\to} \hat{s}_1 \overset{a_2}{\to} \hat{s}_2 \overset{a_3}{\to} \cdots \overset{a_{n-1}}{\to} \hat{s}_{n-1} \overset{a_n}{\to} \hat{s}_n \rangle$. A set P of run sequences on $\mathscr{T}(\mathscr{A})$ obtained by concretizing a counter example $\hat{\rho}$ is also defined as follows using the inverse function h^{-1}:*

Abstraction
Inputs $\mathscr{A}$

/* $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\Rightarrow})$ */
$\hat{S} := L$
$\hat{s}_0 := l_0$
$\hat{\Rightarrow} := \emptyset$
foreach $(l_1, a, g, r, l_2) \in T$ **do**
  $\hat{\Rightarrow} := \hat{\Rightarrow} \cup \{l_1, a, l_2)\}$
end for
return $\hat{M}$

Fig. 3 An abstraction algorithm.



The location E is an error location.

Fig. 4 Example of a timed automaton and its abstract model.

$$P = \{\langle s_0 \xrightarrow{d_0} s_0' \xrightarrow{a_1} s_1 \xrightarrow{d_1} s_1' \xrightarrow{a_2} s_2 \xrightarrow{d_2} \cdots \xrightarrow{a_n} s_n\rangle \mid \bigwedge_{i=0}^{n-1}(s_i \in h^{-1}(\hat{s}_i) \wedge d_i \in \mathbb{R}_{\geq 0} \wedge s_i \overset{d_i}{\Rightarrow} s_i' \wedge s_i' \overset{a_i}{\Rightarrow} s_{i+1})\}.$$

If P has at least one element, we find that the counter example $\hat{\rho}$ is feasible on the original timed automaton. Otherwise we find that $\hat{\rho}$ is spurious.

3.2 Initial Abstraction

Initial Abstraction generates an abstract model $\hat{M}_0$ shown in Sect. 3.1 from a timed automaton $\mathscr{A}_0$. Figure 4 represents an example of a timed automaton and its abstract model obtained by applying Initial Abstraction to the timed automaton.

Figure 3 shows the algorithm of Initial Abstraction.

3.3 Simulation

For an abstract counter example $\hat{\rho} = \langle \hat{s}_0 \xrightarrow{a_1} \hat{s}_1 \xrightarrow{a_2} \hat{s}_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} \hat{s}_n\rangle$, Simulation checks if a set P of the corresponding run sequences on the semantic model is empty or not.

It is difficult to obtain P directly on the semantic model, because P may have infinite of sequences. Therefore, in the algorithm, first we obtain a set Π of sequences of transitions on $\mathscr{A}$ corresponding to $\hat{\rho}$. Then, we check if each element in Π is feasible on the $\mathscr{T}(\mathscr{A})$ or not. If there is an element $\pi \in \Pi$ that is feasible on $\mathscr{T}(\mathscr{A})$, we can conclude that $\hat{\rho}$ is a feasible counter example. On the other hand, if all the elements in Π are infeasible on $\mathscr{T}(\mathscr{A})$, we can conclude that $\hat{\rho}$ is spurious.

The set Π of sequences of transitions on $\mathscr{A}$ corresponding to $\hat{\rho}$ is given as follows;

$$\Pi = \{\langle l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} l_2 \xrightarrow{a_3,g_3,r_3} \cdots \xrightarrow{a_n,g_n,r_n} l_n\rangle \mid \bigwedge_{i=0}^{n} \hat{s}_i = l_i \wedge \bigwedge_{i=1}^{n}(l_{i-1}, a_i, g_i, r_i, l_i) \in T\}.$$

Simulation
Inputs $\mathscr{A}, \pi$

/* $\pi = (l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \cdots \xrightarrow{a_n,g_n,r_n} l_n(l_n = e))$ */
$D_0 := \{0^C\}$
$D_0 := up(D_0)$ /* Any elapsing time */
$D_0 := and(D, I(l_0))$ /* Add Invariant of l_0 */
$succ\_list_0 := (l_0, D_0)$
for $i := 1$ to n **do**
  $succ\_list_i := Succ(\mathscr{A}, succ\_list_{i-1}, (l_{i-1}, a_i, g_i, r_i, l_i))$
  if $succ\_list_i = \emptyset$ **then**
    return $succ\_list$
  end if
end for
return $NULL$ /* The counterexample can be reproduced */

Fig. 5 A simulation algorithm.

Π may have a number of sequences corresponding to $\hat{\rho}$ because there might be several transitions in $\mathscr{A}$ corresponding to the transition $\hat{s}_{i-1} \overset{a_i}{\Rightarrow} \hat{s}_i$ even if $\hat{s}_i$ always corresponds to the single location l_i.

Whether $\pi \in \Pi$ is feasible on the $\mathscr{A}$ is determined by calculating a reachable state set on $\mathscr{A}$ along with π. In this process, when the reachable state set becomes empty, we can conclude that π is infeasible.

In the Definition 3.4, we define the successor state set to be reachable by one action transition followed by arbitrary delay transitions.

Definition 3.4 (Successor State Set)**.** *Given a state set (l_1, D_1) on $\mathscr{T}(\mathscr{A})$ for a timed automaton $\mathscr{A}$ and a transition $e = (l_1, a, g, r, l_2)$, a successor state set $succ((l_1, D_1), e)$ from (l_1, D_1) through the transition e is defined as follows;*

$$succ((l_1, D_1), e) = \{(l', r(\nu) + d) \mid \nu \in D_1 \wedge d \in \mathbb{R}_{\geq 0} \wedge (l_1, \nu) \overset{a}{\Rightarrow} (l_2, r(\nu)) \wedge (l_2, r(\nu)) \overset{d}{\Rightarrow} (l_2, r(\nu) + d)\ \}.$$

Lemma 3.1. *Given a state set (l_1, D_1) on $\mathscr{T}(\mathscr{A})$ of a timed automaton $\mathscr{A}$ and a transition $e = (l_1, a, g, r, l_2)$, a reachable state set $succ((l_1, D_1), e)$ $(= (l_2, D_2))$ satisfies the following property;*

$\forall \nu \in D_2, d \in \mathbb{R}_{\geq 0}.\ (l_2, \nu) \overset{d}{\Rightarrow} (l_2, \nu + d)$ *implies* $\nu + d \in D_2$.

Lemma 3.1 is proved by Definition 3.4 obviously.

From Definition 3.4, a k-th reachable state set from the initial state set (l_0, D_0) is obtained by applying $succ$ function k times like $succ(succ(succ(\ldots succ((l_0, D_0), e_0)\ldots), e_{k-2}), e_{k-1})$. In the rest of this paper, by $succ^k(\pi)$ the k-th reachable state set for π is denoted.

For the sequence π of the length n, π is feasible if $succ^n(\pi) \neq \emptyset$, and π is infeasible if there exists $1 \leq k \leq n$ such that $succ^k(\pi)$ equals $\emptyset$.

Figure 5 represents our Simulation algorithm whose inputs are a timed automaton $\mathscr{A}$ and an element π in Π. It

Succ
Inputs $\mathscr{A}, R = (l_1, D), (l_1, a, g, r, l_2)$

/* To obtain a reachable zone by the given action transition */
$D := and(D, g)$ /* add guards of transitions */
$D := reset(D, r)$ /* reset the clocks */
$D := and(D, I(l_2))$ /* add Invariant of l_2 */
/* To obtain a reachable zone by delay transitions */
$D := up(D)$ /* Any elapsing time */
$D := and(D, I(l_2))$ /* add Invariant of l_2 */
return (l_2, D)

Fig. 6 An algorithm to obtain a successor state set.

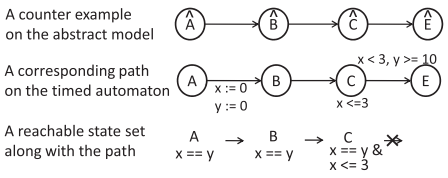


Fig. 7 A simulation process.

checks whether π is feasible on $\mathscr{T}(\mathscr{A})$ or not. The algorithm outputs *NULL* if π is feasible, and otherwise the list of reachable state set *succ_list* along with π, which is used in the Refinement step. Figure 6 shows the algorithm to obtain the successor state set from a given state set and an action transition. The functions $and : c(C) \times c(C) \to c(C)$, $up : c(C) \to c(C)$, and $reset : c(C) \times 2^C \to c(C)$ used in the algorithm are defined as follows. $and(D, g) = \{\nu | \nu \in D \wedge g(\nu)\}$, $up(D) = \{\nu' | \nu \in D \wedge d \in \mathbb{R}_{\geq 0} \wedge \nu \overset{d}{\Rightarrow} \nu'\}$, $reset(D, r) = \{r(\nu) | \nu \in D\}$.

Figure 7 represents a Simulation process in which an abstract counter example on the abstract model of Fig. 4 is checked. As shown in the figure, a reachable state set is represented as a product of a location and a zone. Since any reachable states on the location C don't satisfy the guard condition $x \leq 3 \wedge y \geq 10$ for the transition from C to E, they cannot reach to the error location E. Therefore, we can conclude the counter example is spurious.

3.4 Abstraction Refinement

In the example of Fig. 7, from any state on the location C that is reachable from the initial state, the control cannot move to E due to the guard condition. On its abstract model, however, $\hat{E}$ is reachable because we do not consider the clock constraints on it. This is the cause of the spurious counter example. Generally in such a case, we have to refine the abstract model by dividing the abstract state $\hat{C}$ so that the state set which is reachable from initial state and the state set which is able to move to D become disjoint. Dividing a state space of a timed automaton usually needs subtraction operation on zones. However, zones are not closed under a subtraction operation [10]; therefore, applying such an approach is difficult. In our approach, we transform the transition relation on the timed automaton preserving its equivalence so that the model behaves correctly even if we don't consider the clock constraints.

Refinement
Inputs $\mathscr{A}_i, \pi, succ\_list$

/* $\pi = \langle l_0 \overset{a_1, g_1, r_1}{\longrightarrow} l_1 \overset{a_2, g_2, r_2}{\longrightarrow} \cdots \overset{a_n, g_n, r_n}{\longrightarrow} l_n (l_n = e) \rangle$ */
/* $succ\_list = \langle (l_0, D_0), (l_1, D_1), \cdots, (l_k, D_k) \rangle$,
where (l_j, D_j) represents the j-th reachable state set along with π, and l_k is the last location reachable from the initial state. */
$\mathscr{A}_{i+1} := \mathscr{A}_i$
for $j := succ\_list.length$ downto 1 **do**
  $e_j := (l_{j-1}, a_{j-1}, g_{j-1}, r_{j-1}, l_j)$
  $\mathscr{A}_{i+1} := Duplication(\mathscr{A}_{i+1}, succ\_list_j, e_j)$
    /* Duplication of the Location and Transitions */
  if $IsRemovable(\mathscr{A}_{i+1}, succ\_list_j, e_j)$ **then**
    $\mathscr{A}_{i+1} := RemoveTransition(\mathscr{A}_{i+1}, e_j)$
      /* Removal of Transitions */
    break
  else if $j = 1$ **then**
    $\mathscr{A}_{i+1} := DuplicateInitialLocation(\mathscr{A}_{i+1}, (l_0, D_0))$
      /* Duplicate the initial location and transitions from the initial location */
  end if
end for
$\hat{M}_{i+1} := Abstraction(\mathscr{A}_{i+1}, h)$
return $\hat{M}_{i+1}$

Fig. 8 A refinement algorithm.

DuplicateInitialLocation
Inputs $\mathscr{A}, (l_0, D_0)$

$l'_0 := newLoc()$ /* Generate a new location */
$I(l'_0) := Ineqset(D_0)$ /* A set of inequalities representing D_0 */
$L := L \cup \{l'_0\}$
foreach $(l_0, a, g, r, l) \in T$ **do**
  $(l, D) := Succ(\mathscr{A}, (l_0, D_0), (l_0, a, g, r, l))$
  if $(l, D) \neq \emptyset$ **then**
    /* Duplicate transitions only feasible from (l_0, D_0) */
    $l' := DuplicateOf(\mathscr{A}, (l, D))$
    if $l' = \perp$ **then**
      $T := T \cup \{(l'_0, a, g, r, l)\}$ /* Case (2) of Def.3.9 */
    else
      $T := T \cup \{(l'_0, a, g, r, l')\}$ /* Case (3) of Def.3.9 */
    end if
  end if
end for
$l_0 := l'_0$
return $\mathscr{A}$

Fig. 9 The duplication of the initial location.

Figure 8 represents our abstraction refinement algorithm. In the algorithm, we first apply equivalent transformation to the original model. Next, we generate the refined abstract model by removing clock variables from the transformed model. The transformation algorithm on the original model is composed of three steps; Duplication of Locations, Duplication of Transitions, and Removal of Transitions. Since we have to preserve model equivalence, we impose the restriction for applying removal of transitions.

These transformation steps add or remove some locations and transitions. Therefore, at each step, we have to construct a timed automaton for the step with new locations and transitions as well as invariants. We can discuss equivalence on each of the new timed automaton and the original

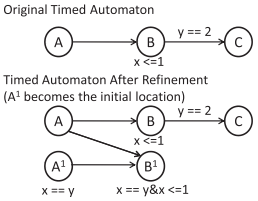


Fig. 10 An example of the algorithm reaches to the initial location.

timed automaton.

The algorithm starts with the reachable last location along with π, and if it cannot apply Removal of Transitions, it traces back π applying the refinement algorithm until being able to apply the removal operation. Finally, if the algorithm reaches to the initial location, it duplicates (l_0, D_0) as $\hat{l}'_0$. Figure 10 shows an example where the algorithm reaches the initial location. Here, at the location A, the algorithm cannot remove a transition from A to B. For such a case, it duplicates the initial location A. We let the new duplicated location A^1 be the initial location. Figure 9 shows the algorithm for the duplication of the initial location.

3.4.1 Duplication of Locations

In Definitions 3.5, 3.6 and 3.7, we give some definitions related to the duplicated locations.

Definition 3.5 (A Parent of a Location)**.** *Let l' be a duplication of the location l, and we call l the parent of l'. The function parent : $L \to L$ is defined as follows;*

$$parent(l) = \begin{cases} l\text{'s parent}, & \text{if } l \text{ has the parent} \\ \bot, & \text{if } l \text{ has no parent.} \end{cases}$$

Definition 3.6 (A Root of a Location)**.** *A root of a location is the eldest ancestor parent of the location. The function root : $L \to L$ is defined as follows;*

$$root(l) = \begin{cases} l, & \text{if } parent(l) = \bot \\ root(parent(l)), & \text{if } parent(l) \neq \bot. \end{cases}$$

In this paper, we use the function *root* to decide whether given locations are duplicated from the same original location. For example, for locations l_1 and l_2 with $root(l_1) = root(l_2)$, we regard l_1 and l_2 are derived from the same original location.

Definition 3.7. *Let $\mathcal{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ be a semantic model of a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$. For a location $l \in L$ and a zone $D \in c(C)$, the function duplicateof : $2^S \times (L \to c(C)) \to L$ is defined as follows;*

$$duplicateof((l, D), I) = \begin{cases} l', & \text{if there exists a location } l' \in L \\ & \quad \text{such that } root(l') = root(l) \\ & \quad \wedge\, D_{I(l')} = D \\ \bot, & \text{otherwise.} \end{cases}$$

For a given state set (l, D), *duplicateof* returns the duplicated location l' of l such that the invariant $I(l')$ corresponds to D. In the definition, $D_{I(l')} = D$ means the equivalence of a zone corresponding to $I(l')$ and D (i.e. $\nu \in D_{I(l')} \iff \nu \in D$).

The algorithm in Fig. 12 implements the function *duplicateof* defined in Definition 3.7. The function $norm_k$ is the k-normalization function defined in [7]. At the line 4 of the algorithm, it checks whether the given zone D is equivalent to $I(l')$ though D is normalized to D_{norm}. If this condition is satisfied for l', we can find that the duplicated location based on the state set (l, D) is already generated, and is l'.

Definition 3.8 (Duplication of Locations)**.** *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ and a path π corresponding to the spurious counter example, suppose that we apply refinement to the k-th location l_k and transition e_k of π. And let $succ^k(\pi)$ equal (l_k, D_k). In this case, we generate a new location l'_k as a duplicate of l_k only if $duplicateof((l_k, D_k), I) = \bot$. Also, we impose a location invariant on l'_k as $I(l'_k) = Ineqset(D_k)$.*

Invariant $I(l')$ for the duplicated location l' is stronger than that $(I(l))$ of the original location l. Therefore, for zone representation of them, $D_{I(l')} \subset D_{I(l)}$ holds.

For semantic states $s = (l, \nu)$ and $s' = (l', \nu')$, we consider s' is the duplicate of s if $l = parent(l')$ and $\nu' = \nu$ hold.

3.4.2 Duplication of Transitions

Suppose that we apply refinement to the k-th location l_k and transition e_k of a path π corresponding to the spurious counter example. In Duplication of Transition, we duplicate the transition e_k and also transitions which is feasible on the reachable states on l_k. In Def. 3.9, we define the transitions to duplicate.

Definition 3.9 (Duplication of Transitions)**.** *For a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ and a path π corresponding to the spurious counter example, suppose that we apply refinement to the k-th location l_k and transition $e_k = (l_{k-1}, a_k, g_k, r_k, l_k)$ of π. And also let l'_k be the duplicate of k-th reachable state set $succ^k(\pi)$ $(= (l_k, D_k))$. Then l'_k equals $duplicateof((l_k, D_k), I)$ $(\neq \bot)$. In Duplicate of Transitions, we duplicate the following transitions;*

- *a duplicate of e_k*

  $$(l_{k-1}, a_k, g_k, r_k, l'_k) \quad (1)$$

- *duplicates of action transitions $e = (l_k, a, g, r, l) \in T$ which are feasible from (l_k, D_k)*

  – *In the case when a duplicate location corresponding to $succ((l_k, D_k), e)$ has not been generated;*

  $$\{\ (l'_k, a, g, r, l) \mid e = (l_k, a, g, r, l) \in T \wedge (l, D) = succ((l_k, D_k), e) \neq \emptyset \wedge duplicateof((l, D), I) = \bot\ \} \quad (2)$$

```
Duplication
Inputs 𝒜, (l_k, D_k), e_k = (l_{k-1}, a_k, g_k, r_k, l_k)
  l'_k := DuplicateOf(𝒜, (l_k, D_k))
  if l'_k = ⊥ then
    /* Duplicate a Location */
    l'_k := newLoc()                          /* Generate a new location */
    L := L ∪ {l'_k}
    D_norm := norm_k(D_k)                     /* k-normalize the zone D_k */
    I( l'_k ) := Ineqset(D_norm)
                              /* A set of inequalities representing D_k */
  end if

  /* Duplicate transitions */
  T := T ∪ {(l_{k-1}, a_k, g_k, r_k, l'_k)}   /* Case (1) of Def.3.9 */
  foreach (l_k, a, g, r, l) ∈ T do
    (l, D) := Succ(𝒜, (l_k, D_k), (l_k, a, g, r, l))
    if (l, D) ≠ ∅ then
                  /* Duplicate transitions only feasible from (l_k, D_k) */
      l' := DuplicateOf(𝒜, (l, D))
      if l' = ⊥ then
        T := T ∪ {(l'_k, a, g, r, l)}         /* Case (2) of Def.3.9 */
      else
        T := T ∪ {(l'_k, a, g, r, l')}        /* Case (3) of Def.3.9 */
      end if
    end if
  end for
  return 𝒜
```

Fig. 11 An algorithm for duplication.

– *In the case when a duplicate location corresponding to* $succ((l_k, D_k), e)$ *has been generated;*

$$\{\ (l'_k, a, g, r, l')\ \mid\ e = (l_k, a, g, r, l) \in T \wedge (l, D) = succ((l_k, D_k), e) \neq \emptyset \wedge l' = duplicateof((l, D), I) \neq \bot\} \quad (3)$$

Expression (1) duplicates a transition e_k (a transition from l_{k-1} to l_k). The duplicated one is a transition from l_{k-1} to l'_k.

Expressions (2) and (3) duplicate a transition e such that its starting location is l_k and can fire from $succ^k(\pi)$. The duplicated one is a transition from l'_k. Expression (3) is for the case that there exists a duplicated location l' such that l' is derived from (l, D) and (l, D) is reachable from $succ^k(\pi)$ via e. For such a case, it duplicates a transition to l' instead of l.

Figure 11 shows both of the duplication algorithms for locations and transitions, because duplication algorithm for transitions depends on the information of that for locations.

Here, we give a lemma about transitions duplicated in the algorithm.

Lemma 3.2. *Given a timed automaton* $\mathscr{A}$ *and a path* π *corresponding to the spurious counter example, suppose that we apply refinement to the k-th location* l_k *and transition* e_k *of* π*, and let* $(l_k, D_k) = succ^k(\pi)$*. The semantic model* $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ *of* $\mathscr{A}$ *satisfies a following property;*

> *There exists a transition* $(s_1, a, s_2) \in \Rightarrow$ *such that* $s_1 \in (l_k, D_k)$*, if and only if there exists a duplicated transition* $(s'_1, a, s'_2) \in \Rightarrow$ *such that* s'_1 *is the duplicate of* s_1*, and* $s'_2 = s_2$ *or* s'_2 *is the duplicate of* s_2*.*

```
DuplicateOf
Inputs 𝒜, (l, D)
  l' := ⊥
  D_norm := norm_k(D)                  /* k-normalize the zone D */
  foreach l_1 ∈ L such that root(l_1) = root(l) do
    if D_{I(l_1)} = D_norm then
      l' := l_1
      break
    end if
  end for
  return l'
```

Fig. 12 An algorithm to check if there is a duplicated location of (l, D) or not.

```
RemoveTransition
Inputs 𝒜, e
  /* e is a transition to remove */
  T := T \ {e}
  return 𝒜
```

Fig. 13 An algorithm for removal of transitions.

```
IsRemovable
Inputs 𝒜, R = (l', D'), e = (l, a, g, r, l')
  R' := Succ(𝒜, (l, D_{I(l)}), e)
                  /* obtain the whole states reachable from l */
  return (R = R')
```

Fig. 14 An algorithm to check if the transition is removable or not.

Proof. From the Expressions (2) and (3) in the Definition 3.9, we duplicate all the transitions which are feasible from the states in (l_k, D_k). Also, for transitions duplicated in the step, the transformed automaton has the original transitions on which the duplication is based. Therefore, $\mathscr{T}(\mathscr{A})$ satisfies the given property. □

3.4.3 Removal of Transitions

Let $e_k = (l_{k-1}, a_k, g_k, r_k, l_k)$ be the k-th action transition in a path π, and $(l_k, D_k) = succ^k(\pi)$ be the k-th reachable state set on π. Here, we also assume that a duplicated location $l'_k = duplicateof((l_k, D_k), I)$ and a duplicated transition $e'_k = (l_{k-1}, a_k, g_k, r_k, l'_k)$ are generated by the latest application of the duplication operations. Here, let us consider that a semantic state $s = (l_k, \nu)$ and its duplicated state $s' = (l'_k, \nu)$ are essentially equivalent (all the enable transitions from the states are equivalent) for all $\nu \in D_k$. Then, if the successor state set $succ((l_{k-1}, D_{I(l_{k-1})}), e_k)$ is equal to (l_k, D_k), it seems that we can substitute e'_k for e_k because $succ((l_{k-1}, D_{I(l_{k-1})}), e'_k)$ is equal to (l'_k, D_k). From this fact we can conclude that even if we remove the transition e_k, the equivalence among the semantic models is preserved. The equivalence is proved in Sect. 4.

We define the condition to remove transitions in Definition 3.10. The function *isRemovable* in the definition is implemented as represented in Fig. 14.

Definition 3.10 (Removable Transitions)**.** *Let* $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ *be a semantic model of a timed automaton* $\mathscr{A} = (A, L, l_0, C, I, T)$*. For a transition* $e = (l_1, a, g, r, l_2) \in T$ *and a state set* $(l_2, D_2) \subset S$*, the function* $isRemovable : T \times 2^S \times (L \to c(C)) \to bool$ *is defined as follows;*

$$isRemovable(e, (l_2, D_2), I) = \begin{cases} \textbf{true}, & if\ succ((l_1, D_{I(l_1)}), e) = (l_2, D_2) \\ \textbf{false}, & otherwise. \end{cases}$$

In our approach, as represented in Fig. 8, we finish the refinement operation when $isRemovable(e_k, succ^k(\pi), I)$ becomes true for the first time. Though we are able to produce a finer model if we continue the refinement operation, we finish the operation at this point to produce a coarser model that is enough to remove the spurious counter example.

We define transitions to remove in Definition 3.11, and in Fig. 13 the algorithm for removal of transitions is represented.

Definition 3.11 (Removal of Transitions)**.** *For a path* π *on a timed automaton* $\mathscr{A} = (A, L, l_0, C, I, T)$ *corresponding to the spurious counter example, let m be a minimum positive integer which satisfies* $succ^m(\pi) = \emptyset$*. For a maximum positive integer* $k \leq m$ *which satisfies* $isRemovable(e_k, succ^k(\pi), I) = true$*, we remove k-th transition* e_k *from T in the removal of transitions, if there exists such an integer k.*

In the rest of this paper, we describe $isRemovable(e_k, succ^k(\pi), I)$ as $isRemovable(\pi, k, I)$ simply.

Lemma 3.3. *If a transition was removed by Removal of Transitions, the same transition will not be generated by Duplication of Transition.*

Proof. Without loss of generality, we assume that a transition $e = (l_1, a, g, r, l_2)$ is removed in the i-th loop.

Here, there is (l_2, D_2) such that $isRemovable(e, (l_2, D_2), I_j)$ is true, and $succ((l_1, D_{I_j(l_1)}), e)$ is equal to (l_2, D_2) by Definition 3.11. By Sect. 3.4.3, there is a duplicated location l'_2 such that $l'_2 = duplicateof((l_2, D_2), I_j))$ and a duplicated transition (l_1, a, g, r, l'_2). $D_2 \subset D_{I_i(l_2)}$ also holds.

We have to consider three cases that the transition e is regenerated in $j(> i)$-th loop. The three cases are corresponding to Expressions (1), (2), and (3) of Definition 3.9. Let assume that a sequence which is corresponding to a counter example detected in the j-th loop, is π_j.

Case Expression (1): In such a case, the $(k-1)$-th location of π_j is l_1. Also the k-th transition is $e_k = (l_1, a, g, r, l_k)(root(l_k) = root(l_2))$, where l_2 equals $duplicateof(succ(\pi_j, k)), I_j)$. However, a set of every reachable state from l_1 via e_k is (l_k, D_2). From the fact $D_2 \subset D_{I_j(l_2)} (= D_{I_i(l_2)}))$, $l_2 = duplicateof(succ(\pi, k)), I_j)$ does not hold.

Case Expression (2) and (3): Let assume that the k-th location of π_j is l_k and $succ(\pi_j, k)$ is equal to (l_k, D_k). Then for a transition $e' = (l_k, a, g, r, l) \in T_j(root(l) = root(l_2))$, It should hold that $l_1 = duplicateof(l_k, D_k)$, and $duplicateof(succ(l_k, D_k), e'), I_j) = l_2$ or $\perp$. From the facts $duplicateof(succ(l_k, D_k), e'), I_j) = l'_2$ and $l'_2 \neq l_2$ hold, which contradicts.

Thus, we can conclude that for any case of Definition 3.9, the transition e will not be duplicated. Therefore, the lemma is proved. □

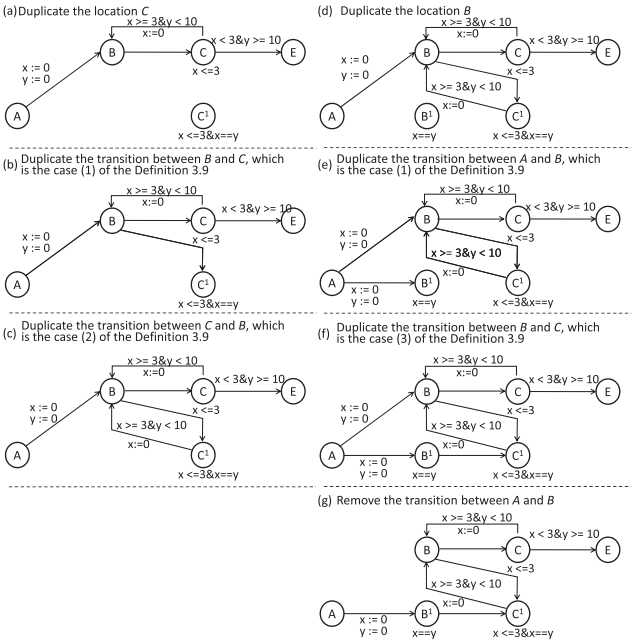


Fig. 15 The refinement process for the path in Fig. 7.

3.4.4 Example

Figure 15 shows a process of Refinement for a counter example in Fig. 7. The last states reachable form the initial location through the path is represented as $(C, x \leq 3 \wedge x == y)$. The algorithm duplicates the location C and obtains a duplicated location C^1. Then it duplicates transitions by Definition 3.9. This process is shown in Fig. 15 (a)–(c). The transition form C to B can fire in the state $(C, x \leq 3 \wedge x == y)$. Therefore it generate a transition from C^1 (Fig. 15 (c)), however it does not duplicate the transition from C to E, because it cannot fire in the state $(C, x \leq 3 \wedge x == y)$.

At this point, if we could remove a transition form B to C, we would have removed the counter example. However, we cannot remove the transition, because the state set of C reachable from any state which satisfies the invariant of B is $(C, x \leq 3)$ and it does not equal $(C, x \leq 3 \wedge x == y)$ which is the reachable state set of C through the path. Instead of this, we perform the algorithm to location B backwards the path (Fig. 15 (d)–(g)).

Figure 15 (f) shows the process of Duplication of Transitions from B to C. Here, the state set of C reachable from the state set $(B, x == y)$, which is the reachable state set on B via the path, is $(C, x \leq 3 \wedge x == y)$. The corresponding location for $(C, x \leq 3 \wedge x == y)$ is already generated as location C^1, hence $duplicateof((C, x \leq 3 \wedge x == y), I) = C^1$. There-

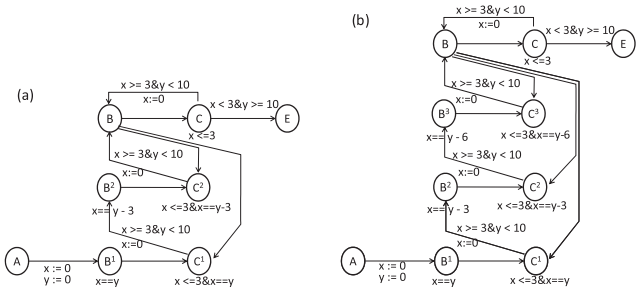


Fig. 16 The timed automata after the second ((a) of the Figure) and third ((b) of the Figure) refinement steps respectively.

fore the transition from B to C is duplicated as a transition from B^1 to C^1.

Figure 15 (g) shows the process of removal of a transition. A state set of B reachable from $(A, true)$ is $(B, x == y)$, which is equal to a state set of B reachable from the initial state set $(A, x == y)$. Therefore a transition e_1 from A to B is removable, because $isRemovable(e_1, (B, x == y), I)$ is true.

Figure 16 (a) and (b) show the obtained timed automata applying second and third refinements, respectively. In the second iteration, a spurious counter example $\langle \hat{A} \xrightarrow{\tau} \hat{B}^1 \xrightarrow{\tau} \hat{C}^1 \xrightarrow{\tau} \hat{B} \xrightarrow{\tau} \hat{C} \xrightarrow{\tau} \hat{E} \rangle$ is detected for the abstract model for the model in Fig. 15 (g) (We assume that unlabelled transition is labelled by τ). For such a spurious counter example, we can obtain a timed automaton in Fig. 16 (b).

In a similar way, the third iteration detects counter example $\langle \hat{A} \xrightarrow{\tau} \hat{B}^1 \xrightarrow{\tau} \hat{C}^1 \xrightarrow{\tau} \hat{B}^2 \xrightarrow{\tau} \hat{C}^2 \xrightarrow{\tau} \hat{B} \xrightarrow{\tau} \hat{C} \xrightarrow{\tau} \hat{E} \rangle$, which is a true counter example. As a result, the algorithm terminates with the report of the counter example.

4. Correctness Proof

Paper [2] gives a theorem on a conservative class of abstractions which attempts to preserve semantics of automata against state reductions under the condition that it checks only a property $\mathtt{AG}\ p$ for a proposition p.

From the theorem, we can derive the following theorem.

Theorem 4.1. *Given a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, $E \subset L$ be a set of error location and $\hat{M}$ be the abstract model. The following statement always holds.*

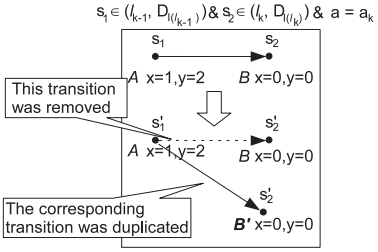
$$\hat{M} \models \mathtt{AG} \bigwedge_{e \in E} \neg \hat{e} \Rightarrow \mathscr{A} \models \mathtt{AG} \bigwedge_{e \in E} \neg e \tag{4}$$

Proof. Let the semantic model of $\mathscr{A}$ be $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$. For a proposition p, if an abstraction function h satisfies the following for every $s \in S$:

$$h(s) \models p \Rightarrow s \models p \tag{5}$$

then $\hat{M} \models \mathtt{AG}\ p \Rightarrow M \models \mathtt{AG}\ p$ holds by Theorem 1 in Paper [2].

Here we assume that p equals $\bigwedge_{e \in E} \neg e$. From the Definition 3.1, $h(l, v)$ equals l. Therefore, $h((l, v)) \models \bigwedge_{e \in E} e \Rightarrow (l, v) \models \bigwedge_{e \in E} e$ obviously holds. As a result, the abstraction function h satisfies the statement 5; Theorem 4.1 is proved. □



The transitions at the top are those of the semantic model of Fig. 4 and at the bottom are those of Fig. 15 (g).

Fig. 17 An example of the case when $s_1 \overset{a}{\Rightarrow} s_2 \notin \Rightarrow'$ in the proof (i) of lemma 4.1.

Next, we prove that the transformation on a timed automaton is equivalent transformation, and also that the abstract model generated by the refinement algorithm is the refined model of the former abstract model.

We prove the equivalence of the transformation by proving that the semantic model of the timed automaton after the transformation is bi-simulation equivalent to the original one.

Lemma 4.1. *Let $\mathscr{A}'$ be a timed automaton model obtained by applying refinement algorithm into a timed automaton $\mathscr{A}$. Then the semantic models of them are bi-simulation equivalent to each other.*

Proof. For semantic models $\mathscr{T}(\mathscr{A}) = (S, s_0, \Rightarrow)$ and $\mathscr{T}(\mathscr{A}') = (S', s'_0, \Rightarrow')$, we define a relation $R \subseteq S \times S'$ as follows, and prove that R is a bi-simulation relation.

$$R = \{(s, s') | s \in S \wedge s' \in S' \wedge ((s = s') \vee (s' \text{ is the duplicate of } s))\}$$

Let us denote a path on the $\mathscr{A}$ corresponding to the spurious counter example used in the refinement step by $\pi = \langle l_0 \overset{a_1, g_1, r_1}{\longrightarrow} l_1 \overset{a_2, g_2, r_2}{\longrightarrow} \cdots \overset{a_n, g_n, r_n}{\longrightarrow} l_n \rangle$, and denote the i-th transition $(l_{i-1}, a_i, g_i, r_i, l_i)$ on π by e_i.

Let m be the minimum positive integer which satisfies $succ^m(\pi) = \emptyset$, and k be the maximum integer within $1 \leq k \leq m$ which satisfies $isRemovable(\pi, k, I) = true$. If there is no such an integer k, we let k be 0. We are going to prove that R is a bi-simulation relation by proving the following properties (*i*) to (iv).

(i) For all $(s_1, s'_1) \in R$ and $a \in A$, if there exists an action transition $s_1 \overset{a}{\Rightarrow} s_2$, there exists a corresponding transition $s'_1 \overset{a}{\Rightarrow}' s'_2$ and $(s_2, s'_2) \in R$ holds.

First, we consider the case when s_1 equals s'_1 holds.

If $s_1 \overset{a}{\Rightarrow}' s_2$ holds, then the property is obviously satisfied.

In the case when $s_1 \overset{a}{\Rightarrow}' s_2$ does not hold, (See Fig. 17)

that is when the transition $s_1 \overset{a}{\Rightarrow} s_2$ is related to the transition e_k, then $k \neq 0$, $s_1 \in (l_{k-1}, D_{I(l_{k-1})})$, $s_2 \in (l_k, D_{I(l_k)})$ and $a = a_k$ hold. In addition to this, $isRemovable(\pi, k, I)$ is true due to Definition 3.11. Also, if $isRemovable(\pi, k, I)$ is true, any successor state from s_1 through e_k is included in (l_k, D_k) due to Definition 3.10, and this implies $s_2 \in (l_k, D_k)$. Also, there is a duplicated location corresponding to (l_k, D_k), and therefore, s_2 has its duplication s'. In addition, by the Duplication of Transitions, the transition $s_1 \overset{a}{\Rightarrow}' s'_2$ is generated as the duplicate of $s_1 \overset{a}{\Rightarrow} s_2$, and $(s_2, s'_2) \in R$ holds.

In the case that holds $s_1 \neq s'_1$, i.e. s'_1 is the duplicate of s_1, by Lemma 3.2 there always exists $s'_1 \overset{a}{\Rightarrow}' s'_2$ as the duplicate transition of $s_1 \overset{a}{\Rightarrow} s_2$. In this case, s'_2 is s_2 itself or the duplicate of s_2. Thus, $(s_2, s'_2) \in R$ holds.

(ii) For every $(s_1, s'_1) \in R$ and $d \in \mathbb{R}_{\geq 0}$, if there exists a delay transition $s_1 \overset{d}{\Rightarrow} s_2$, there exists a corresponding transition $s'_1 \overset{d}{\Rightarrow}' s'_2$ and $(s_2, s'_2) \in R$ holds.

In the case $s_1 = s'_1$, it obviously holds.

In the case $s_1 \neq s'_1$, i.e. s'_1 is the duplicate of s_1, let s_1 and s'_1 be (l, ν) and (l', ν) respectively, which satisfies $l \neq l'$ and $l = parent(l')$. Also, l' is the duplicated location based on a state set (l, D) ($\nu \in D$) which is obtained by the *succ* operation. According to Lemma 3.1, a state set obtained by the *succ* operation closes under delay transitions. Here, although the invariants on l and l' are different, l' is the duplicated location of l. Therefore, the invariant of l' is stronger than that of l and this implies $D \subset D_{I(l)}$. From this fact, we find that for all ν and d , $(l, \nu) \overset{d}{\Rightarrow} (l, \nu{+}d)$ implies $(l, \nu{+}d) \in (l, D)$. Similarly for such ν and d, $(l', \nu{+}d) \in (l', D)$ holds. Therefore, $(l', \nu) \overset{d}{\Rightarrow}' (l', \nu{+}d)$ and $((l, \nu + d), (l', \nu + d)) \in R$ holds.

(iii) For every $(s_1, s'_1) \in R$ and $a \in A$, if there exists an action transition $s'_1 \overset{a}{\Rightarrow}' s'_2$, there exists a corresponding transition $s_1 \overset{a}{\Rightarrow} s_2$ and $(s_2, s'_2) \in R$ holds.

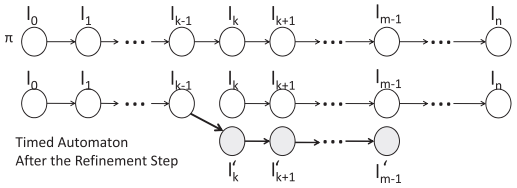
If $s'_1 \overset{a}{\Rightarrow}' s'_2$ is not a duplicated transition, $s'_1 \overset{a}{\Rightarrow} s'_2$ holds obviously.

Otherwise, $s'_1 \overset{a}{\Rightarrow} s'_2$ does not hold. As implied in Lemma 3.2, however, a duplicated transition always has the original one, and therefore, there exists the original transition $s_1 \overset{a}{\Rightarrow} s_2$, and $(s_2, s'_2) \in R$.

(iv) For every $(s_1, s'_1) \in R$ and $d \in \mathbb{R}_{\geq 0}$, if there exists a delay transition $s'_1 \overset{d}{\Rightarrow}' s'_2$, there exists a corresponding transition $s_1 \overset{d}{\Rightarrow} s_2$ and $(s_2, s'_2) \in R$ holds.

This is proved by Lemma 3.1 in a similar manner of the proof of (ii).

From the proof of (i), (ii), (iii) and (iv), R is proved to be a bi-simulation relation. Thus, $\mathscr{T}(\mathscr{A}')$ is bi-simulation equivalent to $\mathscr{T}(\mathscr{A})$. □



The hashed locations, and bold transiitons mean duplicated locations and duplicated transitions respectively.

Fig. 18 An overview of the timed automaton after the refinement step for the path π.

Next, we show that an abstract model generated by applying our algorithm will be a refined model of the previous one.

Lemma 4.2. *Let $\hat{M}'$ be a refined abstract model from $\hat{M}$ by our proposing technique. $\hat{M}'$ is a refined model of $\hat{M}$.*

We prove the lemma by prove that $\hat{M}$ simulates $\hat{M}'$.

Let $\hat{M}$ and $\hat{M}'$ equal $(\hat{S}, \hat{s}_0, \hat{\Rightarrow})$, and $(\hat{S}', \hat{s}'_0, \hat{\Rightarrow}')$, respectively. Then we can define a relation $\hat{R} \subseteq \hat{S}' \times \hat{S}$ as follows and we have only to prove that $\hat{R}$ is a simulation relation.

$$\hat{R} = \{(\hat{s}', \hat{s}) | \hat{s}' \in \hat{S}' \wedge \hat{s} \in \hat{S} \wedge (\hat{s}' = \hat{s}) \vee (\hat{s} = parent(\hat{s}'))\}$$

In this paper, we omit the detailed proof because it is similar to the case (iii) in the proof of Lemma 4.1.

Next, we show that our refinement removes $\hat{\rho}$ for a counter example $\hat{\rho} = \langle \hat{s}_0 \overset{a_1}{\rightarrow} \hat{s}_1 \overset{a_2}{\rightarrow} \cdots \overset{a_n}{\rightarrow} \hat{s}_n \rangle$ on an abstract model $\hat{M}_i$ of i-th iteration.

Here, we show the simple case only. However, other cases are also discussed in a similar way. We assume that path set Π corresponding to $\hat{\rho}$ on timed automaton $\mathscr{A}_i$, has a single element $\pi = \langle l_0 \overset{a_1, g_1, r_1}{\longrightarrow} l_1 \overset{a_2, g_2, r_2}{\longrightarrow} \cdots \overset{a_n, g_n, r_n}{\longrightarrow} l_n \rangle$.

Let integers k and b be the integers defined by Definition 3.11. The algorithm generates duplicated locations $l'_k, l'_{k+1}, \cdots l'_{m-1}$ for the location l_k to l_{m-1}, if these locations have not been generated, and also, it removes the transition $(l_{k-1}, a_k, g_k, r_k, l_k)$ as shown in Fig. 18. It also duplicates related transitions as shown in Fig. 18.

These transitions are duplicated using Expression (3) in Definition 3.9. Each transition from l_{j-1} to l_j is duplicated as a transition from l'_{j-1} to l'_j, where $k \leq j \leq m - 1$.

For abstract model $\hat{M}_{i+1}$ of a timed automaton $\mathscr{A}_{i+1}$ which is obtained after the i-th iteration, we can reach a state $\hat{s}'_{m-1} (= l'_{m-1})$ through a path $\hat{\rho}$. However, we cannot reach a state $\hat{s}_m$ even considering duplicated transitions. Thus, the sprious counter example is removed.

Theorem 4.2 (Correctness)**.** *Our abstraction refinement algorithm refines abstract models correctly.*

Proof. By Lemma 4.1, abstract models before and after the refinement step are bi-simulation equivalent. Also, by Lemma 4.2, an original abstract model simulates its refined abstract model. These lemmas imply our abstraction refinement algorithm refines abstract models correctly. □

Theorem 4.3 (Termination)**.** *The proposed CEGAR algorithm terminates.*

Proof. The proposed algorithm consists of three operations, Duplication of Location, Duplication of Transition, and Removal of Transition.

For a given finite counter example without loop, at least one of the three operations is executed at each iteration. Therefore, our goal is to show that the numbers of duplicated locations, duplicated transitions, and removal of transition are finite. And also we have to show that the algorithm never repeats the process of regeneration of a transition which is once removed.

Duplication of Location duplicates the location for a given location $l \in L$ and zone $D \in c(C)$. In general, a zone on C is finite [7] under the k-normalization. Therefore, the duplicated location also be finite.

Duplication of Transition duplicates a transition. The number of locations including the duplicated ones is finite. And also the number of application of Duplication of Transition is limited to the number of locations. Therefore the number of duplicated transition is also finite.

The transition set which will be removed is subset of whole transitions; it is finite. Lemma 3.3 states that the algorithm never repeats the process of the regeneration. Thus, the algorithm terminates. □

5. Conclusion

This paper proposes a model abstraction technique for timed automata based on the CEGAR algorithm. In general, most CEGAR based algorithms obtain refined abstract models from the previous abstract models by modifying some transformations. In our algorithm, however, the refined model is obtained indirectly; we transform the original timed automaton preserving the equivalence and from it we generate an abstract model by eliminating clock attributes.

This paper gives a formal description and correctness proof of our algorithms.

Future work contains applying subtraction operation [10] in order to divide a bad state into a reachable state and unreachable one instead of duplicating it, during refinement of an abstract model. Comparison its efficiency with the method proposed in this paper is also considered.

Acknowledgments

This work is partially supported by the research grant of the Okawa Foundation, (ref.no. 08-09) and Grant-in-Aid for Scientific Research (C) No.2150036.

Also, this work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by Ministry of Education, Culture, Sports, Science and Technology.

References

[1] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut, "Counterexample-guided abstraction refinement for symbolic model checking," J. ACM, vol.50, no.5, pp.752–794, 2003.

[2] E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning techniques," Proc. 14th Int. Conf. on Computer Aided Verification, Lecture Notes in Computer Science, vol.2404, pp.695–709, 2002.

[3] E.M. Clarke, A. Fehnker, Z. Han, J. Ouaknine, O. Stursberg, and M. Theobald, "Abstraction and counterexample-guided refinement in model checking of hybrid systems," Int. J. Found. Comput. Sci., vol.14, no.4, pp.583–609, 2003.

[4] R. Alur, Techniques for Automatic Verification of Real-Time Systems, PhD thesis, Stanford University, 1991.

[5] R. Alur, C. Courcoubetis, and D.L. Dill, "Model-checking for real-time systems," Proc. 5th Annual Symposium on Logic in Computer Science, pp.414–425, 1990.

[6] S. Das, D.L. Dill, and S. Park, "Experience with predicate abstraction," Proc. 11th Int. Conf. on Computer Aided Verification, Lecture Notes in Computer Science, vol.1633, pp.160–171, 1999.

[7] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," Lectures on Concurrency and Petri Nets, Lecture Notes in Computer Science, vol.3098, pp.87–124, 2004.

[8] F. Wang, K. Schmidt, G.D. Huang, F. Yu, and B.Y. Wang, "Formal verification of timed systems: A survey and perspective," Proc. IEEE, vol.92, no.8, pp.1283–1307, 2004.

[9] G. Behrmann, A. David, and K.G. Larsen, "A tutorial on UPPAAL," Proc. 4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, Lecture Notes in Computer Science, vol.3185, pp.200–236, 2004.

[10] A. David, J. Hakansson, K.G. Larsen, and P. Pettersson, "Model checking timed automata with priorities using DBM subtraction," Proc. 4th Int. Conf. on Formal Modelling and Analysis of Timed Systems, Lecture Notes in Computer Science, vol.4202, pp.128–142, 2006.

[11] H. Nakajima and Y. Kameyama, "Improvement on real-time model checking using abstraction-refinement," Trans. IPSJ, vol.45, no.SIG12 (PRO23), pp.11–24, 2004.

[12] S. Kemper and A. Platzer, "SAT-based abstraction refinement for real-time systems," Proc. Third Int. Workshop on Formal Aspects of Component Software, vol.182, pp.107–122, 2006.

[13] H. Dierks, S. Kupferschmid, and K G. Larsen, "Automatic abstraction refinement for timed automata," Proc. 5th Int. Conf. on Formal Modelling and Analysis of Timed Systems, Lecture Notes in Computer Science, vol.4763, pp.114–129, 2007.

[14] T. Nagaoka, K. Okano, and S. Kusumoto, "Abstraction of timed automata based on counterexample-guided abstraction refinement loop," IEICE Technical Report, SS2007-74, 2008.

Takeshi Nagaoka received the M.I. degree in Computer Science from Osaka University in 2007. He currently belongs in a doctoral course. His research interests include abstraction techniques in model checking, especially timed automaton.

Kozo Okano received the BE, ME, and Ph.D degrees in Information and Computer Sciences from Osaka University, in 1990, 1992, and 1995, respectively. Since 2002 he has been an associate professor in the Graduate School of Information Science and Technology, Osaka University. In 2002, he was a visiting researcher of the Department of Computer Science, University of Kent at Canterbury. In 2003, he was a visiting lecturer at the School of Computer Science, University of Birmingham. His current research interests include formal methods for software and information system design. He is a member of IEEE, and IPS of Japan.

Shinji Kusumoto received the BE, ME, and DE degrees in information and computer sciences from Osaka University in 1988, 1990, and 1993, respectively. He is currently a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of the IEEE, the IEEE Computer Society, IPSJ, and JFPUG.