PAPER

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

Takeshi NAGAOKA†, ***Nonmember***, **Kozo OKANO**†, ***and*** **Shinji KUSUMOTO**†, ***Members***

SUMMARY

Model checking techniques are useful for design of high-reliable information systems. Well-known 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. The proposed technique is based on CEGAR, in which we use a counter example as a guide to refine the model abstracted excessively. Although, in general, the refinement operation is applied to abstract models, our method modifies the original timed automata, and next generates refined abstract models from the modified automata. 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 that 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, 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 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.

Paper[1] proposed an abstraction algorithm called CEGAR (CounterExample-Guided Abstraction Refinement) 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 given specifications, the concrete model also satisfies them.

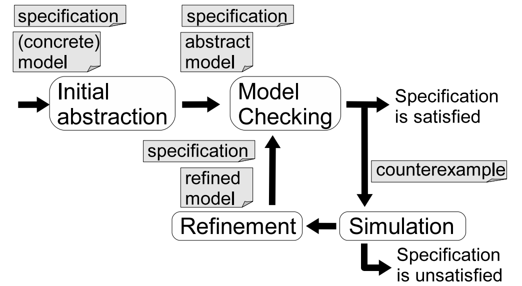


Fig. 1 General CEGAR Algorithm

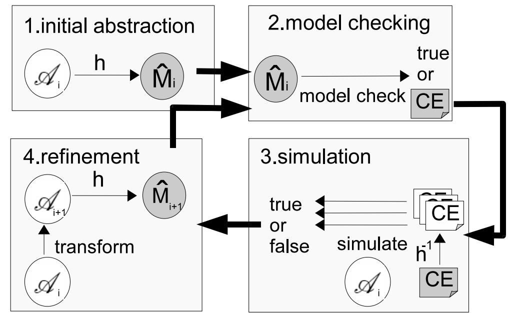


Fig. 2 Our Proposed Algorithm

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, how-

†Graduate School of Information Science and Technology, Osaka University
Machikane-yama 1–3, Toyonaka City, Osaka, 560–8531 Japan

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

As related works, papers[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 Sec. 2, some definitions are described. Sec. 3 gives our CEGAR algorithm and its application to a simple example. Comparison with other related methods is also given. Sec. 4 proves the correctness of the algorithm. Sec. 5 concludes the paper.

2. Preliminaries

In this section, we give definitions of a timed automaton, a region automaton which specifies whole states of a timed automaton with finite clock regions, and others.

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 is 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 a 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;
C: a finite set of clocks;
$l_0 \in L$: an initial location; and
$T \subset L \times A \times 2^{c(C)} \times \mathscr{R} \times L$,
where $2^{c(C)}$ is a set of clock constraints, called guards;
$\mathscr{R} = 2^C$: a set of clocks to reset;
and $I \subset (L \to 2^{c(C)})$: a mapping from locations to clock constraints, called location invariants.

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 can extend the domain of ν into a set of C as follows: $\nu \in \mathbb{R}^C_{\geq 0}$. We define $(\nu + d)(x) = \nu(x) + d$ for $d \in \mathbb{R}_{\geq 0}$. $r(\nu) = \nu[x \mapsto 0], x \in r$ is also defined for $r \in 2^C$. By N, a set of whole ν is denoted.

Definition 2.4 (Semantics of Timed Automaton)**.** *For a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, let a set of whole states of $\mathscr{A}$ be $S = L \times N$. 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 transitions are semantically defined. The first one is called an action transition, while the latter one is called a delay transition.

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

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 $\mathscr{T}(\mathscr{A})$, the semantic model of $\mathscr{A}$ is denoted.*

2.2 Region Automaton

For a given timed automaton $\mathscr{A}$, we can introduce a corresponding clock region $CR(\mathscr{A})$ [4], [5]. By $[u]$, an element (a region) in $CR(\mathscr{A})$ is denoted. For $[u] \in CR(\mathscr{A})$, $g([u])$ and $I([u])$ represent that any point in $[u]$ satisfies a guard g and invariant I, respectively. Also by $r([u])$, applying clock resetting r onto $[u]$ is denoted, where $r([u]) = [u][x \mapsto 0]$, and $x \in r$.

Definition 2.6 (Region Automaton)**.** *A region automaton $\mathscr{A}_r = (A, L_r, l_{r\,0}, T_r)$ of a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ is defined as follows. $L_r \subset L \times CR(\mathscr{A})$, $l_{r\,0} = (l_0, [0^C])$, where $[0^C]$ satisfies $I(l_0)$,*
$T_r \subset L_r \times A \times L_r$,
T_r consists of

$$\begin{aligned}(l, [u]) \overset{a}{\Rightarrow} (l', [v]) \text{ iff } & (l, u) \overset{d}{\Rightarrow} (l, u') \in \mathscr{T}(\mathscr{A}) \text{ for } d \in \mathbb{R}_{\geq 0} \\ & \wedge (l, u') \overset{a}{\Rightarrow} (l', v) \in \mathscr{T}(\mathscr{A}) \text{ for } a \in A \\ & \wedge\; u \in [u] \wedge v \in [v].\end{aligned}$$

2.3 DBM (Difference Bound Matrix)

In [7], [10], a data structure DBM(difference bound matrix) is introduced to manage a set of differential inequalities of two clocks over a finite clock set C.

We represent DBM as a set of some elements in the clock region $CR(\mathscr{A})$. Therefore a set of states of a region

automaton $\mathscr{A}_r = (L_r, l_{r\,0}, T_r, A)$, can be represented in $(l, D) = \{(l, [u]) \mid [u] \in D\}$ using the corresponding DBM D. Paper[7] gives operation functions on DBM, such as up, and and other functions, which represent elapsing time, intersection of time spaces and so on, respectively. There is a minimum set of differential inequalities which can represent DBM D [7]. Such a set is denoted by $Ineqset(D)$. $Ineqset(D)$ can be obtained by reduction operations on DBM. A set of every region which satisfies an invariant $I(l)$ of location l is denoted by (l, D_{Inv}).

2.4 General CEGAR Algorithm

Model abstraction sometimes over-approximates an original model, which may produce spurious counter examples. They 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, we perform model checking on the abstract model. In this step, if the model checker reports that the model satisfies a given specification, 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 a counter example produced 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 Timed Automaton

Our proposed algorithm generates an abstract model $\hat{M}$ from a given timed automaton $\mathscr{A}$ by applying an abstraction function h, and performs model checking on $\hat{M}$. $\hat{M}$ is in fact a finite automaton. If a counter example $\hat{T}$ (represented as a path on the abstract model) is found during model checking, we concretize $\hat{T}$ by applying inverse function h^{-1}.The concretized one is a set of paths. We denote it by T (which is a set of paths on $\mathscr{A}$). At Simulation Step, it checks whether each path in T is feasible on $\mathscr{A}$ or not. If every path in T is infeasible, the next step shall refine the model so that the counter example $\hat{T}$ becomes infeasible. Our algorithm does not directly refine $\hat{M}$ but it modifies $\mathscr{A}$ and then obtains a new abstract mode by applying h to the modified timed automaton. Figure 2 shows flow of our CEGAR algorithm.

The proposed algorithm checks a property $\mathtt{AG} \bigvee_{e \in E} \neg e$, 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 loops. Therefore, hereafter, we assume that counter examples are finite sequences without loops.

3.1 Abstract Model

Definition 3.1 defines the abstraction function h on L_r of a region automaton $\mathscr{A}_r$.

Definition 3.1 (Abstraction Function h)**.** *For a region automaton $\mathscr{A}_r = (A, L_r, l_{r\,0}, T_r)$ of a given timed automaton $\mathscr{A}$, an abstraction function $h : L_r \to \hat{S}$ is defined as follows:*

$$\forall l_{r\,i}, l_{r\,j} \in L_r.\ h(l_{r\,i}) = h(l_{r\,j}) \iff Loc(l_{r\,i}) = Loc(l_{r\,j}),$$

where $Loc : L_r \to L$ is a function which retrieves a location attribute from a state of $\mathscr{A}_r$. The inverse function $h^{-1} : \hat{S} \to 2^{L_r}$ of h is also defined as in a usual manner.

The abstraction function h defined in Definition 3.1 maps any state of L_r which belongs to the same location into the same abstract state. Otherwise they are mapped into the different states. This means that there is a one-to-one correspondence between the location set of $\mathscr{A}$ and the abstract state set $\hat{S}$. Therefore, the abstraction function h can be extended its domain as in Definition 3.2.

Definition 3.2 (Extension of Abstraction Function h)**.** *Abstraction function $h : L \to \hat{S}$ of a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ is defined as follows:*

$$\forall l_i, l_j \in L.\ h(l_i) = h(l_j) \iff l_i = l_j.$$

Similarly, the inverse function $h^{-1} : \hat{S} \to L$ of h is also defined.

Symbols decorated with '$\hat{\ }$' represent those of an abstract model (i.e. $\hat{S}$ represents a state set of an abstract model). Definition 3.3 gives an abstract model $\hat{M}$ of a given timed automaton $\mathscr{A}$ using the abstraction function h defined in Definition 3.2.

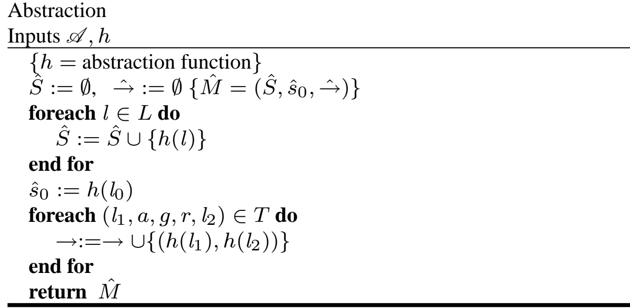
Definition 3.3 (Abstract Model)**.** *An abstract model $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\rightarrow})$ of a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ using the abstraction function h defined in Definition 3.2 is defined as follows:*

- $\hat{S} = \{h(l) \mid l \in L)\}$,
- $\hat{s_0} = h(l_0)$, *and*
- $\hat{\rightarrow} = \{(\hat{l_1}, a, \hat{l_2}) \mid (l_1, a, g, r, l_2) \in T\}$.

Definition 3.4 (Counter Example)**.** *A counter example on $\hat{M}$ is a sequence of states of $\hat{S}$. A counter example $\hat{T}$ of length n is represented in $\hat{T} = \langle \hat{s}_0, \cdots, \hat{s}_n \rangle$.*

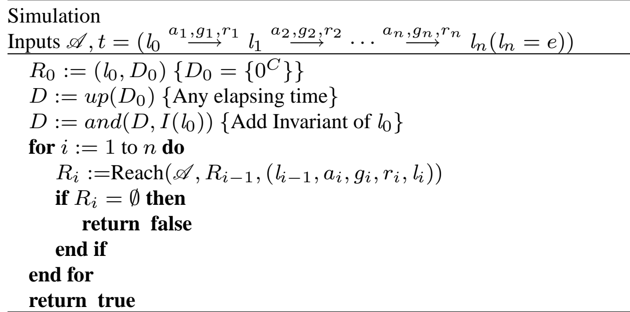
A set T of a run sequences on $\mathscr{A}$ obtained by concretizing a counter example $\hat{T} = \langle \hat{s}_0, \cdots, \hat{s}_n \rangle$, is defined as follows:

$$\begin{aligned} T = \{ & (l_0 \xrightarrow{a_1, g_1, r_1} l_1 \xrightarrow{a_2, g_2, r_2} \cdots \xrightarrow{a_n, g_n, r_n} l_n) \mid \\ & (\ l_i = h^{-1}(s_i) \text{ for } 0 \le i \le n\ ) \wedge \\ & (\ (l_{i-1}, a_i, g_i, r_i, l_i) \in T \text{ for } 1 \le i \le n\ )\}. \end{aligned}$$



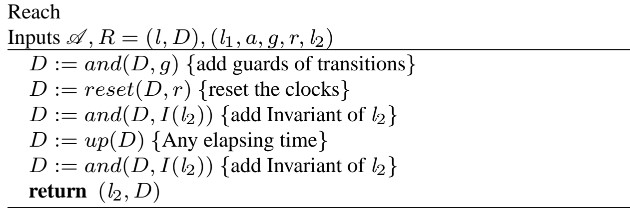
Abstraction
Inputs $\mathscr{A}, h$
$\{h =$ abstraction function$\}$
$\hat{S} := \emptyset, \ \hat{\rightarrow} := \emptyset \ \{\hat{M} = (\hat{S}, \hat{s}_0, \hat{\rightarrow})\}$
foreach $l \in L$ **do**
$\hat{S} := \hat{S} \cup \{h(l)\}$
end for
$\hat{s}_0 := h(l_0)$
foreach $(l_1, a, g, r, l_2) \in T$ **do**
$\hat{\rightarrow} := \hat{\rightarrow} \cup \{(h(l_1), h(l_2))\}$
end for
return $\hat{M}$

Fig. 3 Abstraction



Simulation
Inputs $\mathscr{A}, t = (l_0 \xrightarrow{a_1, g_1, r_1} l_1 \xrightarrow{a_2, g_2, r_2} \ldots \xrightarrow{a_n, g_n, r_n} l_n (l_n = e))$
$R_0 := (l_0, D_0) \ \{D_0 = \{0^C\}\}$
$D := up(D_0)$ {Any elapsing time}
$D := and(D, I(l_0))$ {Add Invariant of l_0}
for $i := 1$ to n **do**
$R_i :=$Reach$(\mathscr{A}, R_{i-1}, (l_{i-1}, a_i, g_i, r_i, l_i))$
if $R_i = \emptyset$ **then**
return false
end if
end for
return true

Fig. 4 Simulation



Reach
Inputs $\mathscr{A}, R = (l, D), (l_1, a, g, r, l_2)$
$D := and(D, g)$ {add guards of transitions}
$D := reset(D, r)$ {reset the clocks}
$D := and(D, I(l_2))$ {add Invariant of l_2}
$D := up(D)$ {Any elapsing time}
$D := and(D, I(l_2))$ {add Invariant of l_2}
return (l_2, D)

Fig. 5 Reach

3.2 Initial Abstraction

Initial Abstraction generates an abstract model $\hat{M}$ from a timed automaton $\mathscr{A}$ using the abstraction function h. Figure 3 shows the algorithm of Initial Abstraction.

3.3 Simulation

For a set T of concretized counter example sequences obtained from $\hat{T}$ on $\hat{M}$, Simulation performs the algorithm in Fig.4 on each sequence $t \in T$. Reachability from the first location of t to the last location of t is checked in Simulation using a procedure Reach in Fig.5. Reach uses some operation functions of DBM. The DBM operation "up" applies time elapsing to the DBM, and "and" imposes a differential inequality to the DBM[7]. In the algorithm, we extend the domain of the operation "and" to clock constraints. When the algorithm in Fig.4 returns false, the counter example $\hat{T}$ is judged as a spurious counter example. By t_{unable}, the concrete path input is denoted. The next step refines the abstract model with t_{unable}.

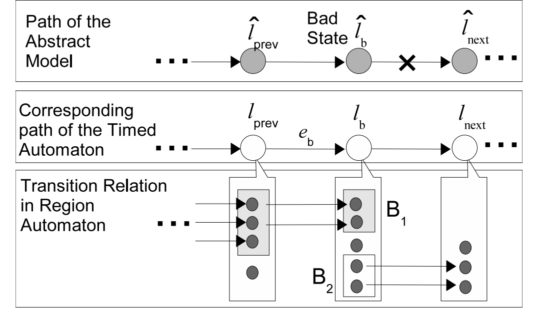


Fig. 6 Counter Example

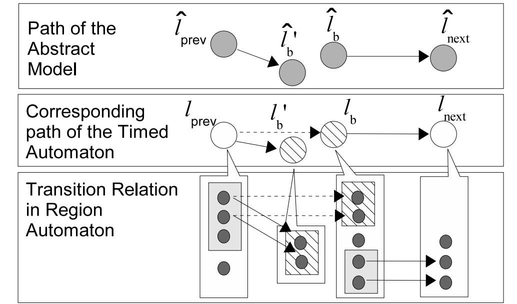


Fig. 7 Refined Model

3.4 Refinement of Abstract Model

In this step, we have to generate a refined abstract model which does not admit the spurious counter example (we call it the spurious CE free model for a given CE). When a counter example is judged as a spurious counter example for a concretized path t_{unable}, there is a Bad State $\hat{l}_b$ in the abstract model (Fig.6). A Bad State $\hat{l}_b$ is the abstract state that the state sets $B_1 (= (l_b, D_1))$ and $B_2 (= (l_b, D_2))$ are merged (mapped into the same state). The state set B_1 has the states which are reachable from the initial state but unreachable to the next location l_{next}. On the other hand, the state set B_2 has the states which are unreachable from the initial state, but reachable to l_{next}. In general, refinement algorithm should divide the state $\hat{l}_b$ into more than two states as the state set B_1 and the state set B_2 are mapped into differential states. Dividing of a state space of a timed automaton usually needs subtraction operation of DBM. However, DBM is not closed under subtraction operation[10]; therefore, applying such an approach is difficult.

We propose another approach, in which we duplicate the state set B_1 on the concrete model instead of dividing a bad state. Also we perform other transformation on the concrete model so that the states in B_2 become unreachable from the initial state obviously (without considering clock constraints). Next, we produce the spurious CE free model by applying the abstraction function to the transformed concrete model.

The algorithm of Refinement in Fig.8 consists of three

sub algorithms, called duplication of states, duplication of transitions, and removal of transitions, shown in Fig.9, 10, and 11, respectively. The state set B_1 is obtained on the way in the Simulation algorithm. At the line 6 in Fig.4, when $R_i = \emptyset$ is true, B_1 corresponds to a previous reachable set R_{i-1}.

In the algorithm 'DuplicateState', we generate the state set $B'_1 = (l'_b, D_1)$ as the duplication of B_1. In the timed automaton level, this transformation generates just a new location l'_b, and the location l'_b has an invariant $Ineqset(D_1)$ which represents D_1.

The algorithm 'DuplicateTransition' generates a transition $(l_{prev}, a, g, r, l'_b)$ as a duplication of the transition (l_{prev}, a, g, r, l_b). Also, we duplicate transitions which are feasible from the states in (l_b, D_1). Due to this transformation, we can establish bi-simulation relations between the states in (l_b, D_1) and their duplications in (l'_b, D_1)[15].

The algorithm 'RemoveTransition' removes the transition (l_{prev}, a, g, r, l_b) if the following condition is satisfied; the state set which is reachable from (l_{prev}, D_{inv}) through (l_{prev}, a, g, r, l_b) equals B_1. In such a case, because we can assume the transition (l_{prev}, a, g, r, l_b) equals $(l_{prev}, a, g, r, l'_b)$, we can remove the transition. Otherwise, we cannot remove the spurious counter example with only one-time application of a Refinement algorithm. In such a case, the following CEGAR loops will find the same spurious counter example, and we apply Refinement algorithm for it again. In Sec.4, we prove that applying Refinement algorithm for the same spurious counter example several times can remove it from the abstract model.

The operation "relation(D, D')" returns a relation between D, D' such as $\langle D \subseteq D', D \supseteq D' \rangle$.

Figure 7 shows a refined model. Dotted arrows in the figure denote transitions which are removed through Refinement algorithm.

For states to duplicate, transitions to duplicate and transitions to remove, the following lemmas hold. Proofs of them are obtained from algorithm straightforward way, and is omitted due to paper space.

Lemma 3.1 (States to Duplicate)**.** *Let $B_1 = (l_b, D_1)$ and duplication of a location l_b be l'_b. A set of states to duplicate, of a region automaton is (l'_b, D_1).*

Duplication of transition duplicates the following kinds of transitions: "transitions from l_{prev} to l_b," and " transitions not only from l_b but also enable from (l_b, D_1)."

Lemma 3.2 (Transitions to Duplicate)**.** *For a region automaton $\mathscr{A}_r = (A, L_r, l_{r\,0}, T_r)$, $B_1 = (l_b, D_1)$, states to duplicate (l'_b, D_1), and a transition $e_b = (l_{prev}, a, g, r, l_b)$ in a counter example, transitions to duplicate of a region automaton is:*

$$T_{r\ d} = \{(l_{prev}, [v]) \overset{a}{\Rightarrow} (l'_b, [v']) \mid \forall (l_{prev}, [v]) \in (l_{prev}, D_{Inv}). \\ \forall (l_b, [v']) \in (l_b, D_1).(l_{prev}, [v]) \overset{a}{\Rightarrow} (l_b, [v']) \in T_r\} \\ \cup \{(l'_b, [v]) \overset{a}{\Rightarrow} (l, [v']) \mid \forall a \in A. \forall (l_b, [v]) \in (l_b, D_1). \\ \forall (l, [v']) \in L_r.(l_b, [v]) \overset{a}{\Rightarrow} (l, [v']) \in T_r\}.$$

Refinement

Inputs $\mathscr{A}_i, h, B_1 = (l_b, D_1), e_b = (l_{prev}, a, g, r, l_b)$

$\{e_b =$ a transition to $l_b\}$
$\mathscr{A}_{i+1} := \mathscr{A}_i$
$\mathscr{A}_{i+1} :=$ DuplicateState($\mathscr{A}_{i+1}, B_1$) {Duplication of States}
$\mathscr{A}_{i+1} :=$ DuplicateTransition($\mathscr{A}_{i+1}, B_1, e_b$)
    {Duplication of Transitions}
$\mathscr{A}_{i+1} :=$ RemoveTransition($\mathscr{A}_{i+1}, B_1$) {Removal of Transitions}
$\hat{M}_{i+1} :=$Abstraction($\mathscr{A}_{i+1}, h$)
return $\hat{M}_{i+1}$

Fig. 8 Refinement

DuplicateState

Input $\mathscr{A}, B_1 = (l_b, D_1)$

$l'_b := newLoc()$ {Generate a new location l'_b}
$L := L \cup \{l'_b\}$
$I(\ l'_b\ ) := Ineqset(D_1)$ {A set of inequalities representing D_1}

Fig. 9 Duplication of States

DuplicateTransition

Inputs $\mathscr{A}, B_1 = (l_b, D_1), e_b = (l_{prev}, a, g, r, l_b)$

$\{e_b =$ a transition to $l_b\}$
$T := T \cup \{(l_{prev}, a, g, r, l'_b)\}$
    {Duplicate a transition e_b to a $BadState$}
foreach $(l_1, a', g', r', l_2) \in T$ such that $l_1 = l_b$ **do**
  if Reach($\mathscr{A}, (l_b, D_1), (l_1, a', g', r', l_2)) \neq \emptyset$ **then**
    $T := T \cup \{(l'_b, a', g', r', l_2)\}$
      {duplicate transitions from l_b only enable from $((l'_b, D_1).)$}
  end if
end for

Fig. 10 Duplication of Transitions

RemoveTransition

Inputs $\mathscr{A}, B_1 = (l_b, D_1), e_b = (l_{prev}, a, g, r, l_b)$

$\{e_b =$ a transition to $l_b\}$
$Prev := (l_{prev}, D_{Inv})$
    {a set of every region satisfying an invariant of l_{prev}}
$R :=$Reach($\mathscr{A}, Prev, e_b$) {obtain regions of l_b reachable from $Prev$}
if $relation(R, B_1) = \langle$**true**, **true**$\rangle$ **then**
    {when $R = B$, $relation(R, B_1)$ returns $\langle$ **true**, **true**$\rangle$.}
  $T := T \setminus \{(l, a, g, r, l_b)\}$
end if

Fig. 11 Removal of Transitions

Lemma 3.3 (Transitions to Remove)**.** *For a region automaton $\mathscr{A}_r = (A, L_r, l_{r\,0}, T_r)$, $B_1 = (l_b, D_1)$, states to duplicate (l'_b, D_1), and a previous location l_{prev} of a location in a counter example, transitions to remove of a region automaton is:*

$$T_{r\ r} = \{(l_{prev}, [v]) \overset{a}{\Rightarrow} (l_b, [v']) \mid \forall (l_{prev}, [v]) \in (l_{prev}, D_{Inv}). \\ (l_{prev}, [v]) \overset{a}{\Rightarrow} (l_b, [v']) \in T_r\}.$$

The algorithm of Removal of Transitions removes transitions only when a set of states reachable from l_{prev} is the same as a set (l_b, D_1) of Bad States. Therefore, for every $(l_{prev}, [v]) \overset{a}{\Rightarrow} (l_b, [v']) \in T_{r\ r}$, $(l_b, [v']) \in (l_b, D_1)$ holds. It means that every transition in $T_{r\ r}$ has its duplication in $T_{r\ d}$.

3.5 Example

Here, we give an example of applying our abstraction method to Light Switch model[7]. The model is shown in Fig12, and it is composed of a switch model (left side of the

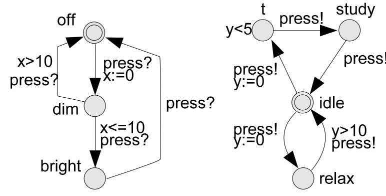


Fig. 12 Light Switch model

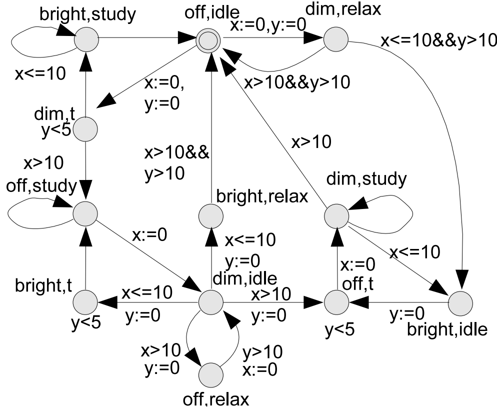


Fig. 13 Parallel composed model

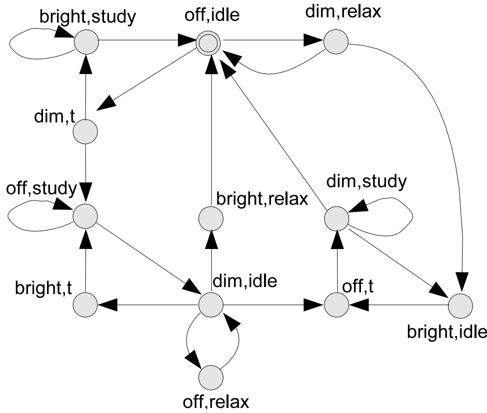


Fig. 14 Initial abstract model

figure) and a user model (right side of the model). Hereafter, we assume that locations $(dim, idle)$ and $(bright, idle)$ of the two models are error locations.

In order to apply our method to these models, first, we have to produce a parallel composition of the models. Figure 13 shows the composition. Transitions with no label in the figure are assumed to be labeled with an action τ. The property which we want to check is:

$$AG\neg((dim, idle) \vee (bright, idle)). \quad (1)$$

For the property (1) and the model of Fig13, the model checker UPPAAL[9] outputs a result of "valid". This means the model of Fig13 satisfies the property (1).

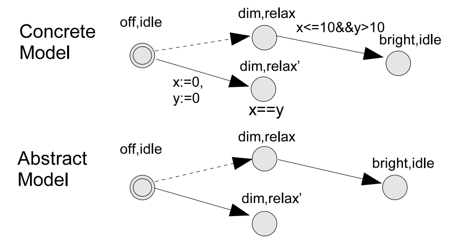


Fig. 15 First refinement

Here, we show an example of applying our abstraction method to the model.

As a first step, we produce an initial abstract model from the parallel composition. In this step, we apply Initial Abstraction which removes clock variables x and y from the composition. Figure 14 shows the initial abstract model.

Next, we perform model checking on the abstract model, and the model checker outputs a counter example $\langle(off, idle), (dim, relax), (bright, idle)\rangle$. This counter example corresponds to a path from $(off, idle)$ to $(bright, idle)$ in the original automaton.

The path in the concrete model in Fig.13 corresponding to this counter example is $(off, idle) \xrightarrow{\tau, true, \{x,y\}} (dim, relax) \xrightarrow{\tau, x\leq 10 \wedge y>10, \emptyset} (bright, idle)$ only. Therefore, we reproduce the path on the concrete model. When we simulate this path on the original automaton, however, a transition from $(dim, relax)$ to $(bright, idle)$ is unable. The reason is as follows; a reachable clock state space of the (bright, idle) always satisfies $x = y$, and it does not satisfy the guard condition $x \leq 10 \wedge y > 10$. Therefore, we can conclude that the counter example is spurious. At the same time, we can obtain the state set $((dim, relax), D_1)$(D_1 is a set of regions which satisfy $x = y$) as B_1, and the transition $((off, idle), \tau, true, \{x, y\}, (dim, relax))$ as e_b.

In the refinement step, first, we duplicate the location $(dim, relax)$ on the timed automaton. (a duplicate of $(dim, relax)$ is denoted by $(dim, relax')$). Please note that we duplicate states only reachable from the initial state, and the reachable state space of $(dim, relax)$ always satisfies $x = y$. Consequently, we have to add an invariant $x = y$ to the duplicated location $(dim, relax')$. Next, we duplicate a transition e_b, and the duplication of this transition is that from $(off, idle)$ to $(dim, relax')$. Also, we duplicate transitions from $(dim, relax)$ except that being unable from the state space which satisfies $x = y$. In this example, we duplicate a transition from $(dim, relax)$ to $(off, idle)$ (and the duplicated transition from $(dim, relax')$ to $(off, idle)$ is depicted in Fig.17), but we don't duplicate a transition from $(dim, relax)$ to $(bright, idle)$ which is infeasible from B_1. Finally, we remove a transition between $(off, idle)$ and $(dim, relax)$. We can remove the transition because there is a corresponding transition $(off, idle)$ to $(dim, relax')$. Figure 15 represents the refinement guided by this counter example. Finally, we produce a refined abstract model from the refined timed automaton.

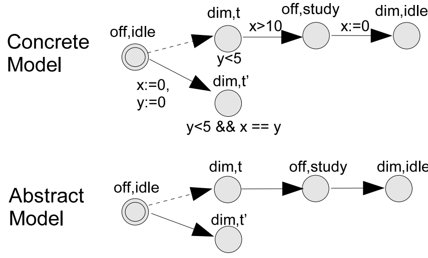


Fig. 16 Second refinement

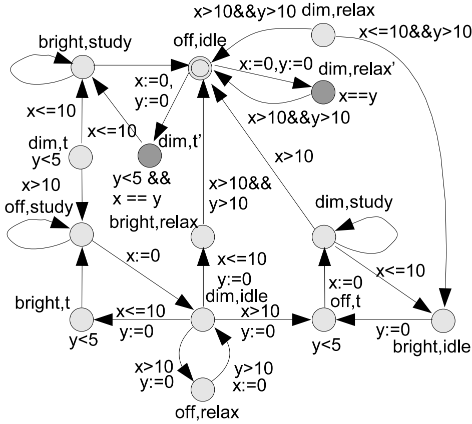


Fig. 17 Timed automaton generated in the final loop

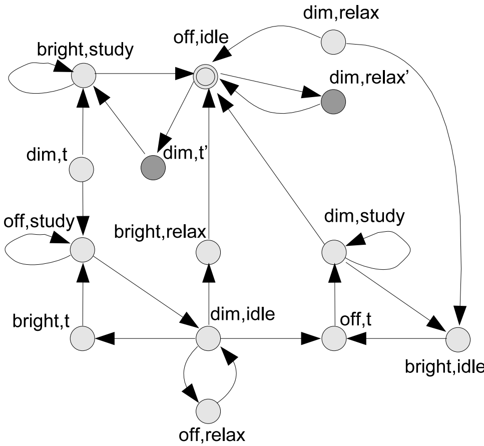


Fig. 18 Abstract model generated in the final loop

After the refinement, we perform model checking again, and we obtain another counter example $\langle(off, idle), (dim, t), (off, study), (dim, idle)\rangle$. For this counter example, Simulation decides it is spurious, and the refinement is performed in the same way. Figure 16 depicts the second refinement.

The third time model checking proves that the model satisfies the property. The timed automaton and abstract model generated in the final loop are presented in Fig17 and Fig18 respectively.

3.6 Related works

As related works, papers[11]–[13] have proposed CEGAR based abstraction techniques for timed automata. In the approach of [11], they perform abstraction by removing all clock variables from timed automata, and refine the abstract model by removing transitions which are always impossible. However, they can remove such transitions only when it is guaranteed that such removal preserves under-approximation, otherwise they have to restore clock variables to the abstract models. The technique of [12] is based on bounded model checking using SAT. In this approach, they refine propositions representing models using spurious counter examples. The technique of [13] limits the model to PLC automata, a sub class of timed automaton. Although these techniques mainly refine abstract models by adding clock variables which have been removed by abstraction, our refinement approach modifies transition relations of models so that the abstract models partially contain real time behavior. The refinement approach of adding clock variables is more effective in that it can remove more spurious counter examples. Adding clock variables, however, may decrease the efficiency of abstraction. On the other hand, because our approach does not add clock variables, we expect our abstraction reduces more state space of the model using traditional techniques on space reduction for ordinal finite automata.

4. Correctness Proof

As mentioned in Section 3, the proposed algorithm checks a property $\texttt{AG} \bigvee_{e \in E} \neg e$, where $E\ (\subset L)$ of a timed automaton $\mathscr{A}$ is a set of error locations of the target system. In other words, we treat only reachability problem. As mentioned in Section 3, any counter example of such a property can be represented in a finite length of sequence without loops. Therefore, we assume that counter examples are finite sequences without loops.

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 $\texttt{AG}\ p$ for a proposition p. From the theorem, we can derive the following theorem.

Theorem 4.1. *For a timed automaton* $\mathscr{A} = (A, L, l_0, C, I, T)$ *and a set* E *of error locations. Let the abstract model and a set of error states of the abstract model be* $\hat{M}$*and* $\hat{E} = \{h(e) \mid e \in E\}$*, respectively. The following statement always holds.*

$$\hat{M} \models \texttt{AG} \bigvee_{\hat{e} \in \hat{E}} \neg \hat{e} \Rightarrow \mathscr{A} \models \texttt{AG} \bigvee_{e \in E} \neg e \qquad (2)$$

Proof. Let a concrete model and its abstract model abstracted by h be M and $\hat{M}$, respectively. For a proposition p, if an abstraction function h satisfies the following for every $s \in S$:

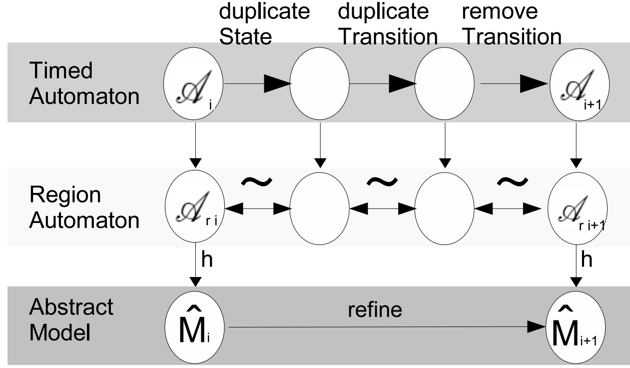


Fig. 19 Relations among models

$$h(s) \models p \Rightarrow s \models p \tag{3}$$

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

Here we assume that $p = \bigvee_{\hat{e} \in \hat{E}} \neg\hat{e}$ for $\hat{M}$, and $p = \bigvee_{e \in E} \neg e$ for $\mathscr{A}$. In addition, an abstraction function defined in Definition 3.2 maps each location in $\mathscr{A}$ to a state $\hat{M}$ and the mapping is one-to-one mapping. Thus, $\forall l \in L.\ h(l) = \hat{e} \iff l = e$ holds. As a result, the abstraction function h satisfies the statement 3; Theorem 4.1 is proved. □

Next, we prove the correctness of our abstraction technique; first, we prove the correctness of our refinement algorithms 'duplicateState,' 'duplicateTransition,' 'removeTransition.' Figure 19 represents the relations among the timed automata generated by each algorithm. Second, we prove that repeating our refinement algorithm can remove a spurious counter example correctly.

Lemma 4.1 (Bi-simulation equivalence among timed automata)**.** *Let denote by $\mathscr{A}_i$ and $\mathscr{A}_{i+1}$ a timed automaton before and after applying $i+1$-th application of Refinement , respectively. $\mathscr{A}_i$ is bi-simulation equivalent to $\mathscr{A}_{i+1}$.*

Proof of Lemma 4.1 is given in Appendix.

For an abstract model $\hat{M}$ and a spurious counter example on $\hat{M}$, we use the term 'the spurious CE free model,' if the refined abstract model of $\hat{M}$ doesn't have the spurious counter example (which may have potentially other spurious CE's).

Lemma 4.2. *For the spurious CE and an abstract model $\hat{M}$, at most n times repetition of Refinement produces the spurious CE free model, where n is the length of the spurious counter example.*

The proof of this lemma is given by showing inductively that for a sub-sequence starting from l_0 to l_k ($1 \leq k \leq n$) of the spurious counter example, at most k times application of the Refinement algorithm refines the abstract model correctly with respect to the sub-sequence. The detail is given in [15].

Lemma 4.3 (Termination)**.** *The CEGAR loop terminates.*

The sketch of proof is as follows. In the worst case, the states of abstract model are divided as fine as product of clock region and locations. They are both finite. One time application of Refinement always divides state space. Consequently, the loop terminates.

Theorem 4.2 (Correctness)**.** *If a counter example is spurious, at most n times repetition of Refinement in Fig.8 produces a spurious CE free model. CEGAR loop will terminate.*

Proof. From Lemma 4.1, Refinement preserves bi-simulation equivalence. From Lemma 4.2, at most n times repetition of Refinement produces a refined spurious CE free model. Lemma 4.3 shows loop's termination. □

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.

As a preliminary experiment, we have applied our abstraction technique to some examples, including Gear Box controller[14], and we have obtained the encouraging result on memory consumption.

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.

References

[1] E M. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut: " Counterexample-guided abstraction refinement for symbolic model checking," Journal of the ACM, vol.50(5), pp752-794, 2003.
[2] E M. Clarke, A. Gupta, J. Kukula, and O. Strichman: "SAT based Abstraction-Refinement using ILP and Machine Learning Techniques," In Proc. of the 14th Int. Conf. on Computer Aided Verification, 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," In Int. Journal of Foundations of Computer Science, vol.14(4), 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 realtime systems," In Proc. of the 5th Annual Symposium on Logic in Computer Science, IEEE, pp.414-425, 1990.
[6] S. Das, D. L. Dill, and S.Park : "Experience with predicate abstraction," In Proc. of the 11th Int. Conf. on Computer Aided Verification, vol.1633, pp.160-171, 1999.

[7] J. Bengtsson, and W .Yi: "Timed Automata: Semantics, Algorithms and Tools," In Lectures on Concurrency and Petri Nets, vol.3098, pp.87-124, 2004.

[8] F. Wang, K. Schmidt, G D. Huang, F. Yu, B Y. Wang: "Formal Verification of Timed Systems: A Survey and Perspective," In Proc. of the IEEE, vol.92, No.8, pp.1283-1307, 2004.

[9] G. Behrmann, A. David, and K G. Larsen: "A Tutorial on UPPAAL," In Proc. of the 4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, 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," In Proc. of the 4th Int. Conf. on Formal Modelling and Analysis of Timed Systems, pp.128-142, 2006

[11] H. Nakajima and Y. Kameyama: "Improvement on Real-Time Model Checking using Abstraction-Refinement (In Japanese)," In Transactions of Information Processing Society of Japan, vol.45, No.SIG12 (PRO23), pp.11-24.

[12] S. Kemper, and A. Platzer: "SAT-based Abstraction Refinement for Real-time Systems," In Proc. of the 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," In Proc. of the 5th Int. Conf. on Formal Modelling and Analysis of Timed Systems, vol.4763, pp.114-129, 2007.

[14] M. Lindahl, P. Pettersson, and W. Yi: "Formal Design and Analysis of a Gear Controller," In Proc. of the 4th International Workshop on Tools and Algorithms for the Constraction and Analysis of Systems, vol.1384, pp.281-297, 1998.

[15] T. Nagaoka, K. Okano, and S. Kusumoto: "Abstraction of Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop," IEICE Technical Report, vol.107, No.505, pp.103-108, 2008.

Appendix: Proof of Lemma 4.1

Proof. For region Automata $\mathscr{A}_r = (A, L_r$ and $l_{r\,0}, T_r)$ and $\mathscr{A}'_r = (A', L'_r, l'_{r\,0}, T'_r)$, we define the following bi-simulation relation $\sim$ recursively.

- For $l_{r\,1} \in L_r$ and $l'_{r\,1} \in L'_r$, if there is a bi-simulation relation $l_{r\,1} \sim l'_{r\,1}$, the following conditions are satisfied.
  for all $a \in A$ and $l_{r\,1} \stackrel{a}{\Rightarrow} l_{r\,2} \in T_r$, there exists a transition $l'_{r\,1} \stackrel{a}{\Rightarrow} l'_{r\,2} \in T'_r$ such as $l_{r\,2} \sim l'_{r\,2}$ holds, and for all $a \in A$ and $l'_{r\,1} \stackrel{a}{\Rightarrow} l'_{r\,2} \in T'_r$, there exists a transition $l_{r\,1} \stackrel{a}{\Rightarrow} l_{r\,2} \in T_r$ such as $l_{r\,2} \sim l'_{r\,2}$ holds.
- For initial states, if $l_{r\,0} \sim l'_{r\,0}$ holds, under the bi-simulation relation $\sim$, $\mathscr{A}_r$ and $\mathscr{A}'_r$ are bi-simulation equivalent.

Let denote by $\mathscr{A}_{r\,i}$ and $\mathscr{A}_{r\,i+1}$ their region automaton for $\mathscr{A}_i$ and $\mathscr{A}_{i+1}$, respectively. In a similar way, $\mathscr{A}_i^1$, $\mathscr{A}^1_{r\,i}$, $\mathscr{A}_i^2$, $\mathscr{A}^2_{r\,i}$, $\mathscr{A}_i^3(=\mathscr{A}_{i+1})$, $\mathscr{A}^3_{r\,i}(=\mathscr{A}_{r\,i+1})$ are defined, where the superfix means a sub algorithm of the Refinement. Therefore the superfixes 1, 2, and 3 stand for after applying Duplication of States, Duplication of Transitions, and Removal of Transition, respectively.

Here, we will prove that $\mathscr{A}_i$ is bi-simulation equivalent to $\mathscr{A}_{i+1}$ by proving bi-simulation equivalence over the each pair of region automata (before and after applying each algorithm). For the state set $B_1 = (l_b, D1)$, we denote the duplicate state set by $B'_1 = (l', D_1)$.

Let $T_{r\,d}$ and $T_{r\,r}$ be a set of transitions to be added in $\mathscr{A}$, a set of transitions to be removed in $\mathscr{A}$, that to be a set of transitions be added in $\mathscr{A}_r$ and that to be removed in $\mathscr{A}_r$, respectively.

i) $\mathscr{A}_{r\,i}$ and $\mathscr{A}^1_{r\,i}$

$\mathscr{A}_{r\,i}$ is obviously bi-simulation equivalent to $\mathscr{A}^1_{r\,i}$ over the bi-simulation relation $\sim$. We omit the proof of it.

ii) $\mathscr{A}^1_{r\,i}$ and $\mathscr{A}^2_{r\,i}$

Let consider $\mathscr{A}_{r\,i} = (L_{r\,i}, l_{r\,i\,0}, T_{r\,i}, A_i)$, and $\mathscr{A}^2_{r\,i} = (L^2_{r\,i}, l^2_{r\,i\,0}, T^2_{r\,i}, A^2_i)$. Obviously, $T^2_{r\,i} = T^1_{r\,i} \cup T_{r\,d}$ holds.

Because $T^1_{r\,i} \subset T^2_{r\,i}$ holds, for $l^1_{r\,i} \in L^1_{r\,i}$ and $l^2_{r\,i} \in L^2_{r\,i}$, such that a bi-simulation relation $l^1_{r\,i} \sim l^2_{r\,i}$ holds, there exists a transition $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T^2_{r\,i}$ where a relation $l^1_{r\,i} \stackrel{a}{\Rightarrow} l^{1\prime}_{r\,i} \in T^1_{r\,i}$ is satisfied for any $a \in A^1_i$. Consequently, the bi-simulation relation $l^{1\prime}_{r\,i} \sim l^{2\prime}_{r\,i}$ also holds.

Let consider converse. For $l^1_{r\,i} \in L^1_{r\,i}$ and $l^2_{r\,i} \in L^2_{r\,i}$ such that the bi-simulation relation $l^1_{r\,i} \sim l^2_{r\,i}$ holds, there exists some transitions such that $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T^2_{r\,i}$ is satisfied. For such transitions, we consider the following cases.

1. The case $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T^1_{r\,i}$ holds.

In this case, $l^1_{r\,i} \stackrel{a}{\Rightarrow} l^{1\prime}_{r\,i} \in T^1_{r\,i}$exists, and the bi-simulation relation $l^{1\prime}_{r\,i} \sim l^{2\prime}_{r\,i}$ holds.

2. The case $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T_{r\,d}$ holds. (a case in which the transition is duplicated)

In this case, $l^2_{r\,i}$ or $l^{2\prime}_{r\,i}$ is in duplicated state set $B'_1 = (l'_b, D_1)$. If $l^2_{r\,i} \in B'_1$ holds, as mentioned in Lemma 3.2, there is a transition which is the source transition for duplication. Therefore, if $l^{2\prime}_{r\,i} \notin B'_1$ holds, $l^{1\prime}_{r\,i} \sim l^{2\prime}_{r\,i}$ also holds. If $l^{2\prime}_{r\,i} \in B'_1$ holds, $l^{2\prime}_{r\,i}$ is a duplicate of $l^{1\prime}_{r\,i}$, and B'_1 is finite. Consequently, from the definition of $\sim$, $l^{1\prime}_{r\,i} \sim l^{2\prime}_{r\,i}$ also holds.

If $l^{2\prime}_{r\,i} \in B'_1$ holds, as mentioned in Lemma 3.2, there is a transition that is the source transition for duplication. Therefore, $l^{2\prime}_{r\,i}$ is a duplicate of $l^{1\prime}_{r\,i}$, and B'_1 is finite. Consequently, from the definition of $\sim$, $l^{1\prime}_{r\,i} \sim l^{2\prime}_{r\,i}$ holds.

For initial states, $l^1_{r\,i\,0} \sim l^2_{r\,i\,0}$ also holds. Therefore, there is the bi-simulation relation $\sim$ between $\mathscr{A}^1_{r\,i}$ and $\mathscr{A}^2_{r\,i}$.

iii) $\mathscr{A}^2_{r\,i}$ and $\mathscr{A}^3_{r\,i}$

Let consider $\mathscr{A}^3_{r\,i} = (L^3_{r\,i}, l^3_{r\,i\,0}, T^3_{r\,i}, A^3_i)$. Obviously, $T^3_{r\,i} = T^1_{r\,i} \setminus T_{r\,r}$ holds.

For $l^2_{r\,i} \in L^2_{r\,i}$ and $l^3_{r\,i} \in L^3_{r\,i}$ such that $l^2_{r\,i} \sim l^3_{r\,i}$ holds, $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T^2_{r\,i}$ holds for any $a \in A^2_i$. For such a transition, we consider the following cases.

1. The case in which $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \notin T_{r\,r}$ holds.

In this case, $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T^3_{r\,i}$ holds. Therefore, $l^3_{r\,i} \stackrel{a}{\Rightarrow} l^{3\prime}_{r\,i} \in T^3_{r\,i}$ exists, and the bi-simulation relation $l^{2\prime}_{r\,i} \sim l^{3\prime}_{r\,i}$ holds.

2. The case in which $l^2_{r\,i} \stackrel{a}{\Rightarrow} l^{2\prime}_{r\,i} \in T_{r\,r}$ holds.

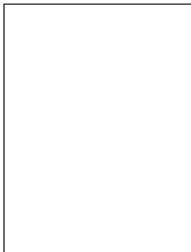
As mentioned in Lemma 3.3, because the transitions to remove have corresponding duplications, there is a duplication of $l^2_{r\ i} \overset{a}{\Rightarrow} l^{2\prime}_{r\ i}$, such that $l^3_{r\ i} \overset{a}{\Rightarrow} l^{3\prime}_{r\ i} \in T^3_{r\ i}$, and $l^{2\prime}_{r\ i} \sim l^{3\prime}_{r\ i}$ holds.

Let consider converse, because $T^3_{r\ i} \subset T^2_{r\ i}$ holds, for all $a \in A^2_i$ such that $l^2_{r\ i} \overset{a}{\Rightarrow} l^{2\prime}_{r\ i} \in T^2_{r\ i}$, there exists a transition $l^2_{r\ i} \overset{a}{\Rightarrow} l^{2\prime}_{r\ i} \in T^2_{r\ i}$. Consequently, the bi-simulation relation $l^{2\prime}_{r\ i} \sim l^{3\prime}_{r\ i}$ holds.

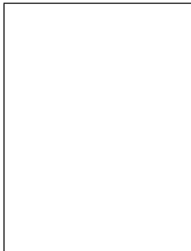
For initial states, $l^2_{r\ i\ 0} \sim l^3_{r\ i\ 0}$ also holds. Therefore, there is the bi-simulation relation $\sim$ between $\mathscr{A}^2_{r\ i}$ and $\mathscr{A}^3_{r\ i}$.

From the facts i), ii) and iii), we can conclude that $\mathscr{A}_{r\ i}$ and $\mathscr{A}_{r\ i+1}$ are bi-simulation equivalent, and also $\mathscr{A}_i$ and $\mathscr{A}_{i+1}$ are. □

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, IEICE of Japan 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, IEICE, and JFPUG.