**Master Thesis**

Title

# Model Abstraction of Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop

Supervisor
Prof. Shinji KUSUMOTO

by
Takeshi NAGAOKA

February 8, 2008

Department of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

# Model Abstraction of Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop

Takeshi NAGAOKA

## Abstract

Recently, for high-reliable information systems, it is strongly desirable to use model checking techniques. Well-known state explosion, however, might occur in model checking of large systems. In particular, in model checking of the real time systems, the number of states of models increases exponentially with the number of clock variables. Such explosion severely limits the scalability of model checking. In order to avoid it, several abstraction techniques are proposed. Some of them are based on CounterExample-Guided Abstraction Refinement (CEGAR) loop technique suggested by E. Clarke *et al.*.

The paper proposes a concrete abstraction technique for timed automata used in model checking of real time systems, and evaluates its efficiency. The proposed technique is also based on CEGAR, in which we use a counter example as a guide to refine the model which is abstracted excessively. Although, in general, the refinement operation is applied to abstract models, the method modifies the original timed automata, and next generates refined abstract models from the modified automata.

Experimental results show the abstraction algorithm can reduce the total memory consumption by at most 80 percent compared to applying model checking without abstraction.

**Keywords**

Model Checking, Timed Automaton, Model Abstraction, CEGAR

# Contents

## List of Figures

## List of Tables

## 1 Introduction

In recent years, there is more and more demand for applying model checking to design of dependable systems. Model checking proves that a given system satisfies specifications by searching a finite transition system exhaustively which describes the system's whole behavior. There is, however, a limitation in scalability including a state explosion. In order to improve the scalability, model abstraction technique becomes important[1, 2, 3].

In verification of real time systems, a timed automaton is used[8, 9], which can describe real time behavior of system . For a timed automaton, real-valued clock constraints are assigned to a state of finite automaton (called location). Therefore, it has an infinite state space which is 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 apply model checking to a finite model. However, the size of the finite model increases exponentially with clock variables; thus an abstraction technique is needed.

Paper[1] proposed a well-organized abstraction algorithm called CEGAR (Counter Example-Guided Abstraction Refinement). The algorithm is used for abstraction of finite models[1, 2], hybrid systems[3], timed automata[14, 15], and other models. In the CEGAR algorithm, we use a counter example produced by a model checker as a guide to refine models which is abstracted excessively. 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.

This paper proposes a model abstraction technique for timed automata based on the CEGAR algorithm. In general, most CEGAR based algorithms[1, 2, 3, 14, 15] 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.

In this paper, we give formal descriptions of our algorithms. Also we prove correctness of our algorithms by proving that the transformation preserves bi-simulation equivalence and that the refined abstract model is the spurious CE free.

As related works, papers[14, 15] proposed CEGAR based abstraction techniques for timed automata. The technique of [14] intends SAT based model checking, and refines propositions which represent models to remove a spurious counter example. The technique of [15] limits the model to PLC automata, a sub class of timed automaton. 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 spurious CE free 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. Sec. 4 proves the correctness of the algorithm. Sec. 6 gives experimental results. Sec. 7 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 a 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;*
*$T \subset L \times A \times 2^{c(C)} \times \mathscr{R} \times L$;*
*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 $\nu$ 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 $\nu$ 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.*

$$\frac{l_1 \xrightarrow{a,g,r} l_2, g(\nu), I(l_2)(r(\nu))}{(l_1, \nu) \overset{a}{\Rightarrow} (l_2, r(\nu))},$$

$$\frac{\forall d' \leq d \quad I(l_1)(\nu + d')}{(l_1, \nu) \overset{d}{\Rightarrow} (l_1, \nu + d)}$$

**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 $\mathscr{T}(\mathscr{A})$, 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 $CR(\mathscr{A})$[5, 6]. In general, a clock region divides a $|C|$-dimensional Euclidean space into finite points, segments, and faces. 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*

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

There is bi-simulation equivalence between a timed automaton $\mathscr{A}$ and its region automaton $\mathscr{A}_r$ [4].

## 2.3 DBM (Difference Bound Matrix)

In [9, 13], a data structure DBM is introduced to represent a convex space in $|C|$ -dimensional Euclidean space, where $C$ is a set of clock variables. DBM is a set of differential inequalities on two clock variables, and represents a state space which satisfies all inequalities over it (the state space is called a zone). DBM represents these set of inequalities as a $|C_0| \times |C_0|$ matrix, where $C_0 = C \cup \{ \mathbf{0} \}$, and **0** is a special variable which means a constant value $0$. A $(i, j)$-th entry ($D_{i\ j}$) of the matrix means a differential inequality of $x_i - x_j$ for $x_i, x_j \in C_0$. Suppose there is an inequality $x_i - x_j \preceq n$ for $\preceq \in \{ < . \leq \}$, the $(i, j)$-th entry $D_{i\ j}$ is represented by $(n, \preceq)$. Also, when $x_i - x_j$ is unbounded, the entry $D_{i\ j}$ is represented by $\infty$. In addition, the upper bound and lower bound of $x_i$ itself are indicated by $D_{0,i}$ and $D_{i,0}$ respectively.

As an example of DBM, let's consider a zone which satisfies following constraint.

$$x - \mathbf{0} < 20 \wedge y - \mathbf{0} \leq 20 \wedge x - y \leq -10 \wedge y - x \leq 10 \wedge \mathbf{0} - z < 5.$$

When we represent this zone as DBM, variables $\mathbf{0}, \ x, \ y, \ z$ are numbered with $0, \ 1, \ 2,$ $3$ respectively in the matrix. DBM which represents the zone of the constraint is given by (1).

$$D = \begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) & (5, <) \\ (20, <) & (0, \leq) & (-10, \leq) & \infty \\ (20, \leq) & (10, \leq) & (0, \leq) & \infty \\ \infty & \infty & \infty & (0, \leq) \end{pmatrix}. \tag{1}$$

DBM is also represented as a set of some elements in the clock region $CR(\mathscr{A})$. Therefore a state 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[9] 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 represents DBM $D$ [9]. Such a set is denoted by $c(D)$. $c(D)$ 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 causes spurious counter examples which are not actually counter examples in the original model.

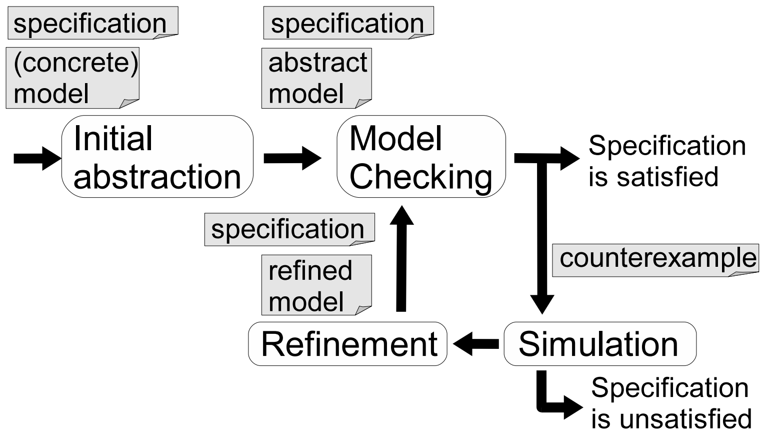

Figure 1: General CEGAR Algorithm

Paper[1] gives an algorithm called CEGAR (Counterexample-Guided Abstraction Refinement) shown in Figure 1.

In the algorithm, at the first step (called Initial Abstraction), it over-approximates the original model. Next, we perform model checking to the abstract model. In this step, if the model checker proofs 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 proofs the model does not satisfy the specification, however, we have to check a counter example produced by it whether it 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 report it to the user and 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.

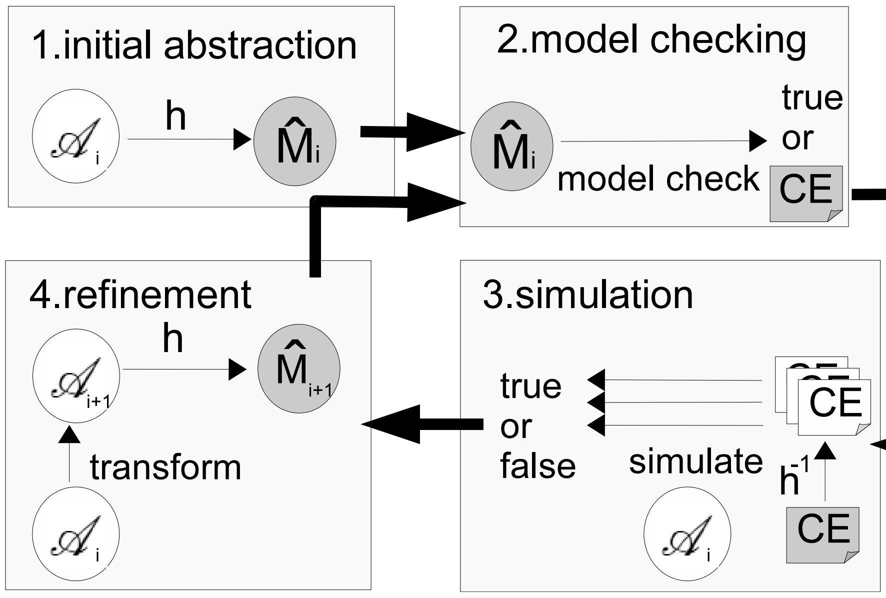

Figure 2: Our Proposed Algorithm

## 3 Proposing CEGAR Algorithm

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 $\hat{M}$. If a counter example $\hat{T}$ (represented as a path on the abstract model) is found while model checking, it concretizes $\hat{T}$ 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 refines $\mathscr{A}$ and then obtains a new abstract mode by applying $h$ to the refined timed automaton. Figure 2 shows flow of our CEGAR algorithm.

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. 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.*

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 $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\to})$ 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:*

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

**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 $\hat{T} = \langle \hat{s}_0, \cdots, \hat{s}_n \rangle$.*

```
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
    $\rightarrow := \rightarrow \cup \{(h(l_1), h(l_2))\}$
  end for
  return $\hat{M}$
```

Figure 3: Abstraction

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

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

## 3.2 Initial Abstraction

Initial Abstraction generates an abstract model $\hat{M}$ from a timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$ 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. When the algorithm in Fig. 4 returns false, the counter example $\hat{T}$ is judged as a spurious counter example.

```
Simulation
Inputs $\mathscr{A}, (l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \cdots \xrightarrow{a_n,g_n,r_n} l_n (l_n = e))$
  $R_0 := (l_0, D_0)$ {$D_0 = \{0^C\}$}
  $D := up(D_0)$ {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
```

Figure 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 $(l_2, D)$
```

Figure 5: Reach

## 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, there is a Bad State $\hat{l}_b$ which has a corresponding state set $B_1 = (l_b, D_1)$ reachable from the initial state but unreachable to $l_{next}$, and another state set $B_2 = (l_b, D_2)$ unreachable from the initial state but reachable to $l_{next}$, are merged (mapped into the same state) as in Fig. 6.

In general, refinement algorithm should divide state $\hat{l}_b$ into more than two states as state $B_1$ and state $B_2$ 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 Subtract operation[13], so applying such an approach is difficult.

We proposes another approach, in which it duplicates state $B_1$ in the concrete model and also performs other transformation on the concrete model. Applying the abstraction function to the transformed concrete model produces a new refinement abstract model where a state mapped from $B_2$ is unreachable (refer in Fig. 7).

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.

Here, we gives definitions of states to duplicate, transitions to duplicate, and transitions to remove.

**Definition 3.5** (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 defined as $(l'_b, D_1)$.*

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 $(l_b, D_1)$.”

**Definition 3.6** (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 previous location $l_{prev}$ of a location $l_b$ in a counter example, transitions to duplicate of a region automaton is defined as follows:*

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

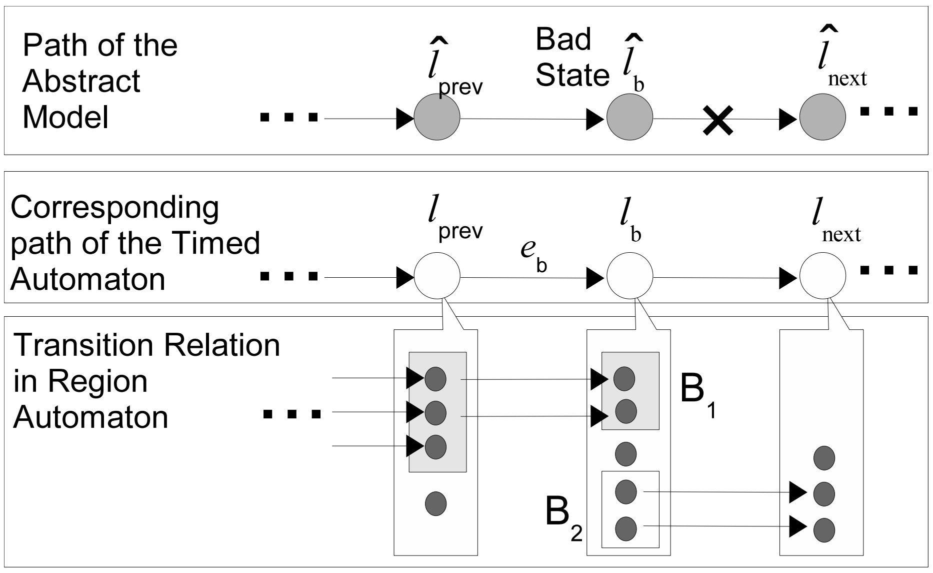
Path of the Abstract Model
$\hat{l}_{prev}$
Bad State $\hat{l}_b$
$\hat{l}_{next}$
Corresponding path of the Timed Automaton
$l_{prev}$
$e_b$
$l_b$
$l_{next}$
Transition Relation in Region Automaton
$B_1$
$B_2$

Figure 6: Counter Example

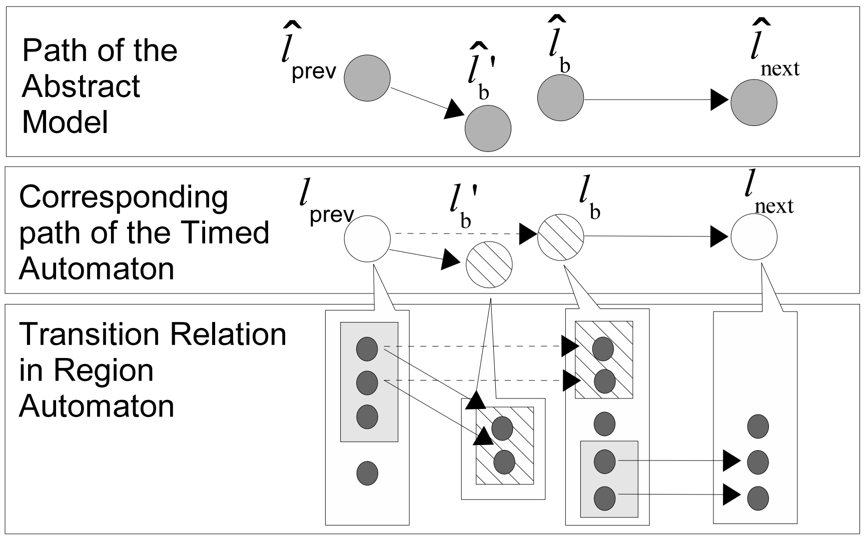
Path of the Abstract Model
$\hat{l}_{prev}$
$\hat{l}_b'$
$\hat{l}_b$
$\hat{l}_{next}$
Corresponding path of the Timed Automaton
$l_{prev}$
$l_b'$
$l_b$
$l_{next}$
Transition Relation in Region Automaton

Figure 7: Refined Model

```
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($\mathscr{A}_{i+1}, B$) {Duplication of States}
    $\mathscr{A}_{i+1} :=$ DuplicateTransition($\mathscr{A}_{i+1}, B, e_b$)
            {Duplication of Transitions}
    $\mathscr{A}_{i+1} :=$ RemoveTransition($\mathscr{A}_{i+1}, B$) {Removal of Transitions}
    $\hat{M}_{i+1}$ :=Abstraction($\mathscr{A}_{i+1}, h$)
    **return** $\hat{M}_{i+1}$
```

Figure 8: Refinement

```
DuplicateState
Input $\mathscr{A}, B_1 = (l_b, D_1)$
    $l_b' := newLoc()$ {Generate a new location $l_b'$}
    $L := L \cup \{l_b'\}$
    $I(\ l_b'\ ) := c(D_b)$ {A set of inequalities representing $D_b$}
```

Figure 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( $\mathscr{A}, (l_b, D_b), (l_1, a', g', r', l_2)) \neq \emptyset$ **then**
            $T := T \cup \{(l_b', a', g', r', l_2)\}$
                {duplicate transitions from $l_b$ only enable from (($l_b'$, $D_b$).)}
        **end if**
    **end for**
```

Figure 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
```

Figure 11: Removal of Transitions

**Definition 3.7** (Transitions to Remove)**.** *For a region automaton* $\mathscr{A}_r = (A, L_r, l_{r\ 0}, T_r)$*,* $B_1 = (l_b, D_d)$*, 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 defined as follows:*

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

.

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

We give an example of applying our abstraction method to Light Switch model[9]. The model is shown in Fig12, and it is composed of a switch model (left side of the figure) and a user model (right side of the model). Hereafter we assume locations $(dim, idle)$ and $(bright, idle)$ of the two models as error locations.

In order to apply our method to these models, first, we have to produce a parallel composition of the models. Figure13 shows the composition. The property which we want to check is:

$$AG\neg((dim, idle) \vee (bright, idle)). \tag{2}$$

When we check the property (2) on the model of Fig13 using the model checker UPPAAL[10, 11, 12], it outputs a result of "valid". This means the model of Fig13 satisfies the property (2).

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 in which we remove clock variables $x$ and $y$ from the composition. Figure14 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.

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.

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 $(dim, relax')$. Also, we duplicate transitions from $(dim, relax)$ except that being unable from the state space which satisfies $x = y$. Next, we remove a transition between $(bright, idle)$ and $(dim, relax)$. We can remove the transition because there is a corresponding transition $(bright, idle)$ to $(dim, relax')$. Figure15 represents the refinement guided by this counter example. Finally, we produce a refined abstract model from the refined timed automaton.

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. Figure16 depicts the second refinement.

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

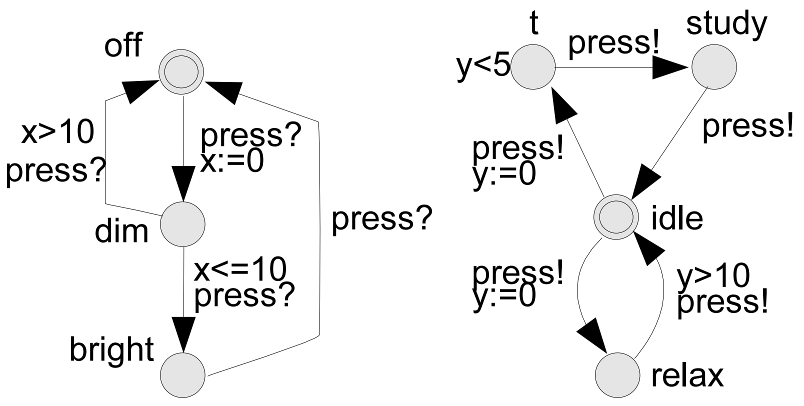

Figure 12: Light Switch model

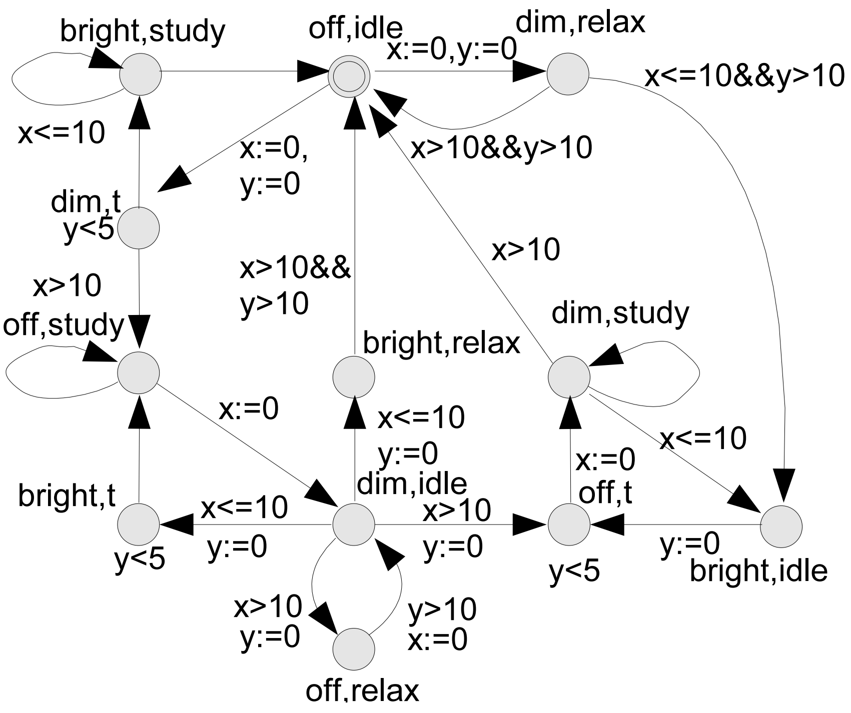

Figure 13: Parallel composed model

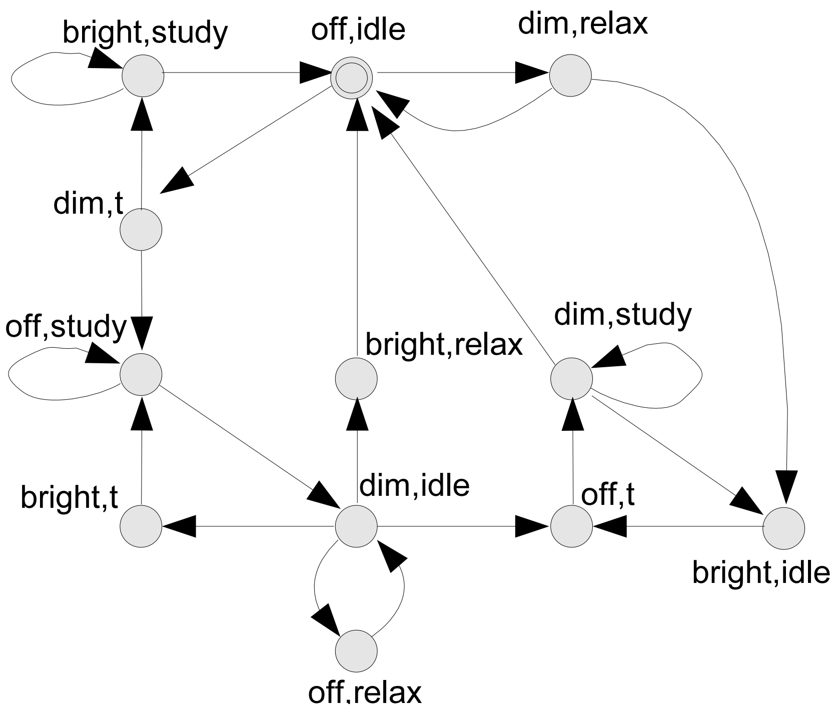

Figure 14: Initial abstract model

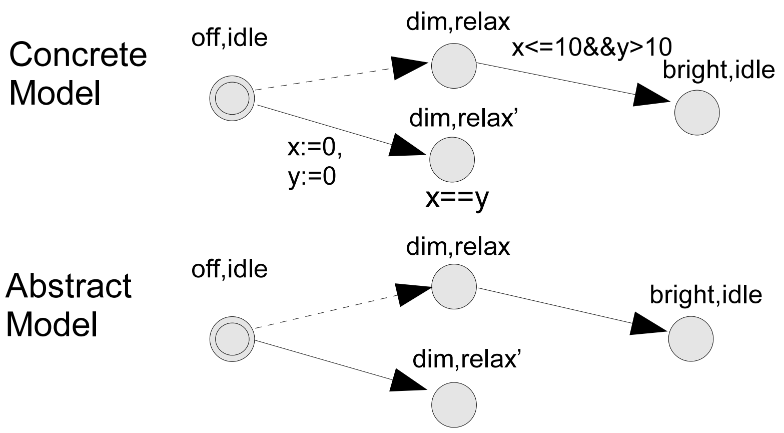

Figure 15: First refinement

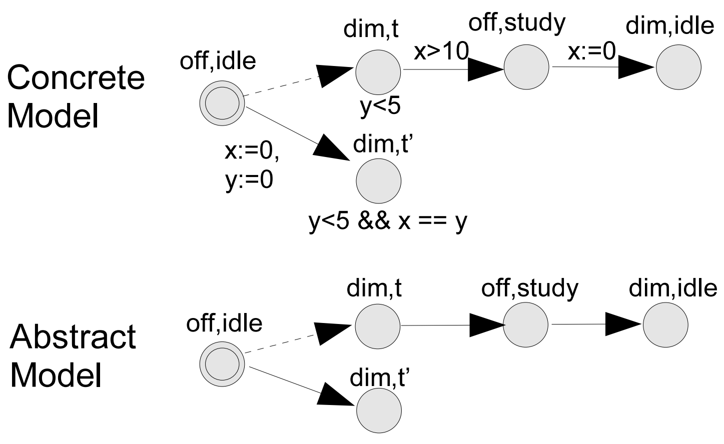

Figure 16: Second refinement

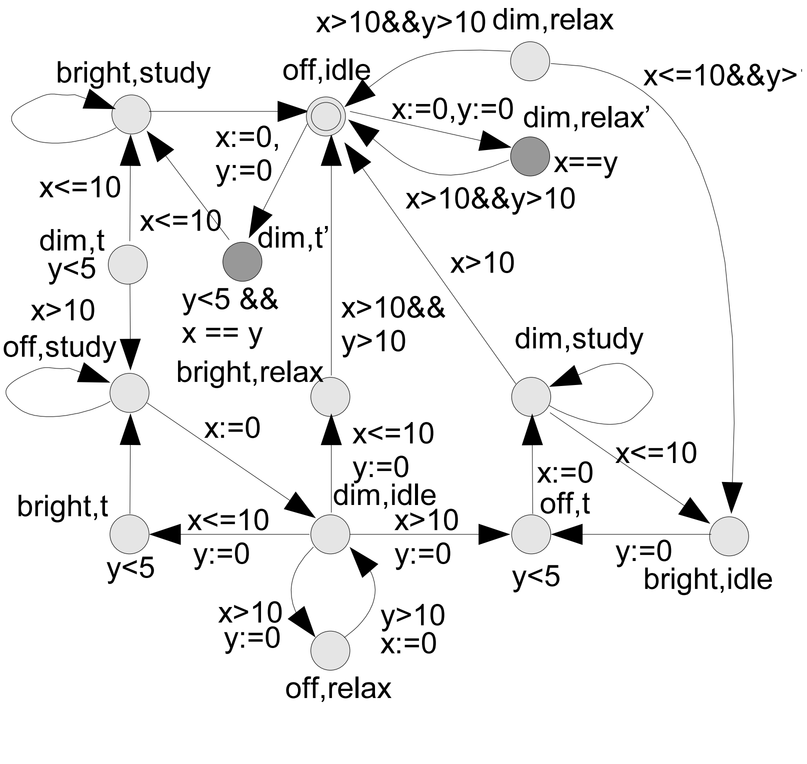

Figure 17: Timed automaton generated in the final loop

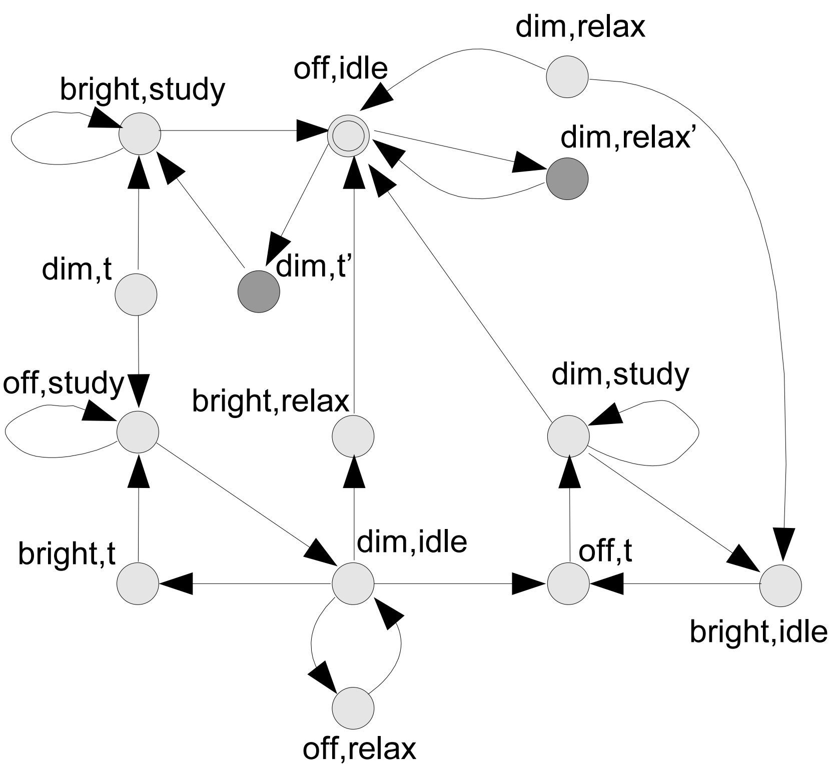

Figure 18: Abstract model generated in the final loop

## 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.

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 $\mathscr{A}$ 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.*

$$\hat{M} \models \texttt{AG} \bigvee_{\hat{e} \in \hat{E}} \neg \hat{e} \Rightarrow \mathscr{A} \models \texttt{AG} \bigvee_{e \in E} \neg e \tag{3}$$

*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 $s \in S$:

$$h(s) \models p \Rightarrow s \models p \tag{4}$$

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, $h(l) = \hat{e} \iff l = e$ holds. As a result, the abstraction function $h$ satisfies the statement 4; Theorem 4.1 is proved. □

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

*Proof.* 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, $\mathscr{A}_i^1$, $\mathscr{A}_{r\ i}^1$, $\mathscr{A}_i^2$, $\mathscr{A}_{r\ i}^2$, $\mathscr{A}_i^3 (= \mathscr{A}_{i+1})$, $\mathscr{A}_{r\ i}^3 (= \mathscr{A}_{r\ i+1})$ 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.

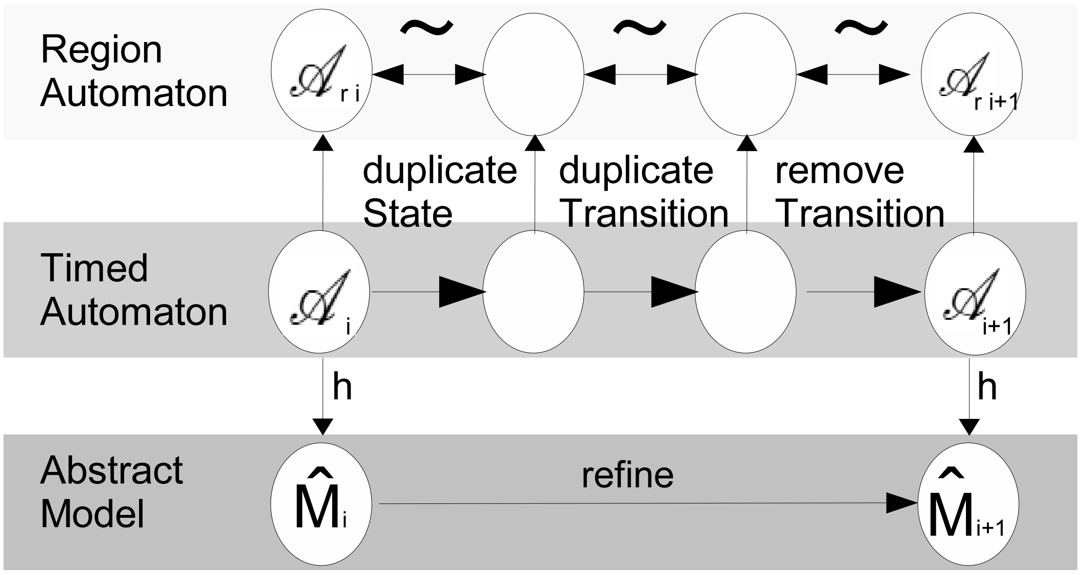

Figure 19: Relations among models

We will prove that $\mathscr{A}_i$ is bi-simulation equivalent to $\mathscr{A}_{i+1}$ by proving bi-simulation equivalence over the corresponding region automata. For $l_b$, let $l'_b$ be a duplicated state. For a set $D_1$ of regions which associates to a location to duplicate, a set of states in $\mathscr{A}_r$ will be $(l_b, D_1)$, and $(l'_b, D_1)$. Let $T_{r\ d}$ and $T_{r\ r}$ 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 $\mathscr{A}^1_{r\ i}$

Let's consider $\mathscr{A}_{r\ i} = (L_{r\ i}, l_{r\ i\ 0}, T_{r\ i}, A_i)$ and $\mathscr{A}^1_{r\ i} = (L^1_{r\ i}, l^1_{r\ i\ 0}, T^1_{r\ i}, A^1_i)$. From the assumption, $l_{r\ i\ 0} = l^1_{r\ i\ 0}$, $T_{r\ i} = T^1_{r\ i}$, $A_i = A^1_i$ and $L^1_{r\ i} = L_{r\ i} \cup (l'_b, D_1)$ hold.

The initial state$l_{r\ i\ 0} = l^1_{r\ i\ 0}$, and $T_{r\ i} = T^1_{r\ i}$. So, there is no transition to the duplicated state set $(l'_b, D_1)$ in $\mathscr{A}^1_{r\ i}$. Thus, there is bi-simulation equivalence between $\mathscr{A}_{r\ i}$ and $\mathscr{A}^1_{r\ i}$.

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

For $\mathscr{A}^2_{r\ i} = (L^2_{r\ i}, l^2_{r\ i\ 0}, T^2_{r\ i}, A^2_i)$, obviously $L^2_{r\ i} = L^1_{r\ i}$ and $l^2_{r\ i\ 0} = l^1_{r\ i\ 0}$, $A^2_i = A^1_i$ hold. $T^2_{r\ i} = T^1_{r\ i} \cup T_{r\ d}$ also holds.

We show that for every $[v] \in D_1$, a state $(l_b, [v])$ and a state $(l'_b, [v])$ have a bi-simulation equivalence relation. When there exists a transition $(l_b, [v]) \overset{a}{\Rightarrow} (l, [v'])$ , as defined in definition 3.6, the corresponding transition $(l_{b'}, [v]) \Rightarrow (l, [v'])$ is generated. Also, when there exists a transition $(l'_b, [v]) \overset{a}{\Rightarrow} (l, [v'])$, there must be an original transition $(l_b, [v]) \overset{a}{\Rightarrow} (l, [v'])$. Thus, we proved the first goal.

Thus, the concrete bi-simulation equivalence relation $\sim$ between $l^1_{r\ i} \in L^1_{r\ i}$ and $l^2_{r\ i} \in L^2_{r\ i}$ is defined as follows:

$$l^1_{r\ i} \sim l^2_{r\ i} \quad \Longleftrightarrow \quad \begin{array}{l} l^1_{r\ i} = l^2_{r\ i} \texttt{ or} \\ l^2_{r\ i} \texttt{ is duplication of } l^1_{r\ i} \end{array} \tag{5}$$

For the initial states, $l^1_{r\ i\ 0} \sim l^2_{r\ i\ 0}$ holds. A transition set $T^1_{r\ i}$ satisfies $T^1_{r\ i} \subset T^2_{r\ i}$. For each transition in $T^1_{r\ i}$, thus, there is a corresponding transition in $T^2_{r\ i}$. Suppose that $l^1_{r\ i} \sim l^2_{r\ i}$ and $l^1_{r\ i} \overset{a}{\Rightarrow} l^{1\prime}_{r\ i}$. Then there exists a transition $l^2_{r\ i} \overset{a}{\Rightarrow} l^{2\prime}_{r\ i}$ and $l^{1\prime}_{r\ i} \sim l^{2\prime}_{r\ i}$. Let consider converse. For each transition in $T^2_{r\ i}$, there is the corresponding transition in $T^1_{r\ i}$. Please note that for a transition in $T_{r\ d}$, there exists the original transition. Suppose that $l^1_{r\ i} \sim l^2_{r\ i}$ and $l^2_{r\ i} \overset{a}{\Rightarrow} l^{2\prime}$. Then there exists a transition $l^1_{r\ i} \overset{a}{\Rightarrow} l^{1\prime}_{r\ i}$ and $l^{1\prime}_{r\ i} \sim l^{2\prime}_{r\ i}$.

Therefore, $\mathscr{A}^1_{r\ i}$ and $\mathscr{A}^2_{r\ i}$ are bi-simulation equivalent.

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

Let's consider $\mathscr{A}^3_{r\ i} = (L^3_{r\ i}, l^3_{r\ i\ 0}, T^3_{r\ i}, A^3_i)$. Obviously $L^3_{r\ i} = L^2_{r\ i}$, $l^3_{r\ i\ 0} = l^2_{r\ i\ 0}$ and $A^3_i = A^2_i$ hold. $T^3_{r\ i} = T^1_{r\ i} \setminus T_{r\ r}$ also holds.

The case when the algorithm in Fig. 11 does not perform any removal of transitions is trivial. $\mathscr{A}^2_{r\ i}$ is equivalent to $\mathscr{A}^3_{r\ i}$, thus also holds the relation $\sim$.

Otherwise, in other words, in the case of removal of a transition, from Definition 3.7, each element in $T_{r\ r}$ has its duplication in $T_{r\ d}$. Thus, even if the transition is removed, $\sim$ is also preserved between $(l_{prev}, [v]) \in (l_{prev}, D_{Inv})$ of $\mathscr{A}^2_{r\ i}$ and $(l_{prev}, [v]) \in (l_{prev}, D_{Inv})$ of $\mathscr{A}^3_{r\ i}$. Thus each state of $L^2_{r\ i}$ and that of $L^3_{r\ i}$ satisfy the relation defined in (5). In a similar way of case ii), $\mathscr{A}^2_{r\ i}$ and $\mathscr{A}^3_{r\ i}$ are bi-simulation equivalent.

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

**Lemma 4.2.** *At most $n$ times repetition of Refinement yields the spurious CE free model, where $n$ is the length of the spurious counter example.*

*Proof.* Let $\mathscr{A}$, $\mathscr{A}_r$ and $\hat{M}$ be a timed automaton, its region automaton and its abstract model, respectively. For a counter example $\hat{T} = \langle \hat{s}_0, \hat{s}_1, \cdots, \hat{s}_n \rangle$, where $\hat{s}_n$ is an abstract state obtained by reducing the error location, let consider one of the corresponding sequences $t = (l_0 \xrightarrow{a_1,g_1,r_1} l_1 \xrightarrow{a_2,g_2,r_2} \cdots \xrightarrow{a_n,g_n,r_n} l_n)$ to $\hat{T}$ on $\mathscr{A}$, where $l_n$ is error location. Let $R_i$ be a set of reachable $i$-th states along with the sequence $t$, and $UR_i$ be that of unreachable $(= (l_i, D_{Inv}) \setminus R_i)$.

We prove that "for sub-sequence starting from $l_0$ to $l_k (1 \leq k \leq n)$ of $t$, by applying at most $k$ times repetition of Refinement yields that it is reachable to an abstract state corresponding to $R_k$ but unreachable to an abstract state corresponding to $UR_k$." (*)

Let duplicated location from $R_i$ be $l'_i$. Let the abstract state of $l'_i$ be $\hat{s'_i} (= h(l'_i))$.

**i)** $k = 1$

$R_0 = (l_0, D_{Inv})$ holds. A set of reachable states from $(l_0, D_{Inv})$ through a transition $(l_0, a_1, g_1, r_1, l_1)$ is in fact $R_1$ from the definition of $R_i$. Therefore, Refinement duplicates $R_1$, which is a location $l'_1$ and Refinement also removes a transition from $l_0$ to $l_1$. In the obtained abstract model, it is reachable to only $\hat{s'_1}$ corresponding to $R_1$, and it is unreachable to a state $h(l_1)$ corresponding to $UR_1$.

**ii)** $k \geq 2$

As inductive assumption, we assume that at most $k-1$ times repetition of Refinement yields that it is reachable to an abstract state corresponding to $R_{k-1}$ but unreachable to an abstract state corresponding to $UR_{k-1}$.

Let $R'_k (\supseteq R_k)$ be a set of reachable states from $(l_{k-1}, D_{Inv})$. If $R_k = R'_k$, then in a similar way as $k = 1$, applying one more Refinement leads to the goal.

Let consider when $R_k \subset R'_k$ holds. A transition from $l_{k-1}$ to $l_k$ cannot be removed because $UR_k$ is reachable from $(l_{k-1}, D_{Inv})$. In such a case, from the inductive assumption, we can obtain the refined abstract model, in which an abstract state corresponding to $R_{k-1}$ is reachable but $UR_{k-1}$ is not. Let $l'_{k-1}$ and $l'_k$ be duplicated locations of $R_{k-1}$ in $k-1$-th time-Refinement and $R_k$ in $k$-th time-Refinement, respectively. Adding transition from $l'_{k-1}$ to $l'_k$ improves the model so that it is reachable to only a state corresponding to $R_k$.

From (i) and (ii), statement (*) is proved.

If the counter example is spurious, it is unreachable from $R_{n-1}$ to error state $(l_n, D_{Inv})$ in $M$. Similarly, in $\hat{M}$, it is unreachable from $\hat{s'}_{n-1}$ to$\hat{s}_n$. Thus the lemma is proved. □

**Theorem 4.2** (Correctness)**.** *If a counter example is spurious, at most $n$ times repetition of Refinement in Fig. 8 yields a spurious CE free model.*

*Proof.* From Lemma 4.1, Refinement preserves bi-simulation equivalence. From Lemma 4.2, at most $n$ times repetition of Refinement yields a refined spurious CE free model. □

## 5 A tool based on the algorithm

In [20], a tool for evaluating the proposed abstraction is implemented. The inputs of the tool are an UPPAAL model file written in xml format and also a query file which describes specifications, and it outputs the result of model checking. In the tool, the algorithms of Initial Abstraction, Simulation, and Abstraction Refinement are implemented, and it performs model checking by calling a model checking command in UPPAAL, ‘verifyta’. When a given model does not satisfy a given specification, Execution of this command produces a counter example as a file. In the tool, to interpret the output file, it calls a command ‘tracer’ provided by UPPAAL parser library.

The tool is implemented in Java, and details of it are presented in [20]

## 6 Experimental Results

This section shows experimental results of applying the proposed abstraction to some examples using the tool, and also evaluates effectiveness of the abstraction mainly from the view points of space consumption. The experiments are executed in the following environment.

| | | |
|---|---|---|
| OS | : | Fedora 7 |
| CPU | : | AMD Athlon(tm) 64 Processor 3400+ 2.2GHz |
| Memory | : | 930MB |
| UPPAAL | : | version 4.0.6 |
| Eclipse | : | version 3.3.1.1 |
| JDK | : | version 1.5.0_14 |

### 6.1 Aims of Experiments

In the experiments, we evaluate how the proposing abstraction saves memory consumption. Specifically, we compare memory consumption in performing UPPAAL model checking with the proposing abstraction with that without it.

### 6.2 Results of Experiments

Here, results of applying to examples "Fischer's mutual execution protocol"[11, 18], and "Gearbox Controller"[19] are presented. In [20], other results are shown.

#### 6.2.1 Fischer's mutual execution protocol

The model of Fischer's mutual execution protocol is composed of several number of same processes. The experiments are executed on the models of $2$ processes to $8$ processes, and a requirement specification is that more than two processes do not stay in the critical section at the same time. Table 1 shows results of these experiments.

The columns of 'original' and 'composition' represent results of performing model checking to the original model and parallel composition of it, respectively, and they are the result without abstraction. Also, the columns of 'abstract' represent results of applying

Table 1: Result of Fischer's mutual execution protocol

| | | | original | | composition | | abstraction | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Proc | location | clock | time | mem | time | mem | time | mem | loop | dup |
| 2 | 16 | 2 | 0.10 | 2.81 | 0.10 | 2.82 | 0.74 | 2.82 | 5 | 10 |
| 3 | 64 | 3 | 0.10 | 2.81 | 0.10 | 2.81 | 2.44 | 2.82 | 13 | 30 |
| 4 | 256 | 4 | 0.10 | 2.81 | 0.20 | 5.51 | 10.9 | 21.6 | 25 | 60 |
| 5 | 1024 | 5 | 0.10 | 2.81 | 0.81 | 51.0 | 69.4 | 48.1 | 41 | 100 |
| 6 | 4096 | 6 | 0.10 | 2.81 | 5.26 | 125 | 662 | 95.0 | 61 | 150 |
| 7 | 16384 | 7 | 0.30 | 37.2 | 57.8 | 427 | 12571 | 293 | 85 | 210 |
| 8 | 65536 | 8 | 1.11 | 38.4 | N/A | N/A | N/A | N/A | N/A | N/A |

abstraction. The columns of 'time' represent total execution time (sec), and those of 'mem' represent the maximum memory consumption during model checking (MB). The columns of 'loop', 'dup' represent a count of loop, and a number of duplicated location, respectively. In the experiment for the eight processes, results are not obtained because 'verifyta' cannot handle the model which has more than $2^{16}$ locations.

The results of this experiments show the proposed abstraction can reduce the memory consumption from applying model checking without abstraction in the cases of more than five processes. In particular, in the case of seven processes, it can reduce it about 30 percent.

### 6.2.2 Gearbox Controller

The model of Gearbox Controller is composed of five processes, and has five clock variables. Paper[19] gives 14 requirement specifications to verify for the model, and the proposed abstraction method can verify 5 specifications of them (specifications (7) to (11) in [19]). Table 2 shows results of experiments. The column of 'spec' means a specification verified in its experiment.

The results of the second experiment show more effectiveness of the abstraction. For all specifications, the abstraction reduces the memory consumption more than 80 percent.

## 6.3 Discussion

Let consider results of our algorithm (see abstraction columns of in Table 1 and Table 2). In Fischer's mutual execution protocol, the size of memory which the tool consumes is

Table 2: Result of Gear Box Controller

| | original | | composition | | abstraction | | | |
|---|---|---|---|---|---|---|---|---|
| spec | time | mem | time | mem | time | mem | loop | dup |
| (7) | 0.10 | 2.82 | 0.82 | 50.0 | 5.51 | 8.71 | 3 | 8 |
| (8) | 0.10 | 2.81 | 0.81 | 50.0 | 5.35 | 8.71 | 3 | 8 |
| (9) | 0.10 | 2.78 | 0.81 | 50.0 | 7.47 | 8.67 | 5 | 18 |
| (10) | 0.10 | 2.81 | 0.82 | 50.0 | 7.63 | 8.84 | 5 | 49 |
| (11) | 0.10 | 2.82 | 0.81 | 50.0 | 7.30 | 8.67 | 5 | 22 |

about 70% of that of parallel composite version (parallel), whereas in Gearbox Controller, The rate becomes about 20%.

The reason why the memory reduction rate of Fischer's mutual execution protocol is lower than that of GearBox Controller are (1) state explosion caused by parallel composition and (2) the number of clock guards is very small; the guard expression $x > k$, $x \leq k$ only exists on the transition to Critical Session location.

In such a case, normalization operation [9] for DBM reduces the size of clock state space, consequently, UPPAAL can effectively reduce memory size. On the other hand, Gearbox Controller uses a lot of clock constraints, which increases the size of total state space.

Unfortunately, without parallel composition scheme (original) has the best results. Our approach needs parallel composition which increases the memory size (see columns 'original' and 'composition' in Table 1 and Table 2). In the original version, the size is very small, because UPPAAL creates the total state space on-the-fly from parallel procedure presentation and also it uses Partial Order Reduction[16] techniques to reduce the size of total state space, especially in Fischer's mutual execution protocol.

## 6.4 Complexity

Here, the computational complexity and space complexity of our algorithm (Initial Abstraction, Simulation and Refinement) are given.

For $\mathscr{A} = (L, l_0, T, I, C, A)$, let $n = |C|$.

**Initial Abstraction** From the algorithm of Fig3, the computational complexity is $O(|L| + |T|)$ and the space complexity is also $O(|L| + |T|)$

**Simulation** Let a length of counter example be $l$. In the algorithm $Reach$ in Fig5, operational functions $up$ and $and$ are used. The computational complexity of $up$ and $and$ are $O(n)$ and $O(n^2)$, respectively. Therefore, the computational complexity of $Reach$ is $O(n^2)$. Also, because a new DBM is generated in the algorithm $Reach$, the space complexity is $O(n^2)$ . Because the algorithm $Reach$ is called $l$ times in Simulation, its computational complexity and space complexity are $O(l \times n^2)$ and $O(l \times n^2)$, respectively.

**Abstraction Refinement** The computational complexity of the algorithms $Duplicate$ $State$, $DuplicateTransition$, $RemoveTransition$ is $O(n^3)$, $O(|T|)$, $O(n^2)$. Therefore that of the algorithm Refinement becomes $O(n^3 + |T|)$. Also, the space complexity of these algorithms are $O(n^2)$, $O(|T|)$, and $O(1)$. Thus, that of the algorithm Refinement is $O(n^2 + |T|)$.

## 7 Conclusion

This paper proposes a model abstraction technique for timed automata based on the CEGAR algorithm. In general, most CEGAR based algorithmsobtain 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 formal descriptions of our algorithms, and also correctness proof of our algorithms by proving that the transformation preserves bi-simulation equivalence and that the refined abstract model is the spurious CE free.

The future work will be extensions of our algorithm. First, we want to handle integer variables used in UPPAAL timed automata[10, 11, 12]. To abstract the state space over integer variables, we are considering applying predicate abstraction[7] to it. Second, we want to extend a range of specification formula for model checking.

In addition, we are considering that when we refine an abstract model, we apply a Subtraction operation[13] to divide a bad state into a reachable state and unreachable one instead of duplicating it, and also compare its efficiency with the method proposed in this paper.

## Acknowledgements

This work could be achieved owing to a great deal of helps of many individuals.

First, I would like to thank my supervisor Professor Shinji Kusumoto of Osaka University, for his continuous support, encouragement and guidance of the work.

I would like to express my sincere gratitude to Associate Professor Kozo Okano of Osaka University for his adequate guidance, valuable suggestions and discussions throughout this work.

I'm grateful to Assistant Professor Yoshiki Higo of Osaka University for his helpful comments and suggestions.

I'm thankful to Mr. Hironori Ashikari, an undergraduate student of Osaka University, for his valuable help in this work especially Section 6.

Finally, I would like to thank the all members of Kusumoto Laboratory of Osaka University for their helpful advice.

## References

[1] E M. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut: “Counterexample-guided Abstraction Refinement,” In Proc. of the 12th Int. Conf. on Computer Aided Verification, vol.1855, pp.154-169, July, 2000.

[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, July, 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.4, No.4, 2003.

[4] E M. Clarke, O. Grumberg, and D A. Peled: “Model Checking,” MIT Press, 2000.

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

[6] R. Alur, C. Courcoubetis, and D. L. Dill: “Model-checking for real-time systems,” In Proc. of the 5th Annual Symposium on Logic in Computer Science, pp.414-425, IEEE Computer Society Press, 1990.

[7] 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.

[8] R. Alur: “TimedAutomata,” In Proc. of the 11th Int. Conf. on Computer Aided Verification, vol.1633, pp.688, July, 1999.

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

[10] UPPAAL
http://www.uppaal.com/

[11] 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, pp200-236, 2004

[12] K G. Larsen, P. Pettersson, and W. Yi: In Springer International Journal of Software Tools for Technology Transfer, vol.1(1-2), pp.134-152, 1997.

[13] 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, 2002

[14] S. Kemper, and A. Platzer: "SAT-based Abstraction Refinement for Real-time Systems," In Third International Workshop on Formal Aspects of Component Software, 2006.

[15] 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.

[16] J. Bengtsson, B. Jonsson, J. Lilius and W. Yi: "Partial Order Reductions for Timed Systems," In Proc. of the 9th Int. Conf. on Concurrency Theory, vol.1466, pp.41-49, 1998.

[17] J. Hakansson, and P. Pettersson: "Partial Order Reduction for Verification of Real-Time Components," In Proc. of the 5th Int. Conf of Formal Modelling and Analysis of Timed Systems, vol.4763, pp211-226, 2007.

[18] J. Bengtsson, K G. Larsen, F. Larsson, P. Pettersson, and W. Yi: "UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems," Hybrid Systems 1995, pp232-243.

[19] 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.

[20] H. Ashikari: "Implement of Model Abstraction of Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop and its Evaluation on Effectiveness," Bachelor Thesis, School of Engineering Science, Osaka University, Feb, 2008.

[21] T. Nagaoka, K. Okano, and S. Kusumoto: “Abstraction of Extended Timed Automata for UPPAAL Based on Counterexample-Guided Abstraction Refinement Loop (in Japanese),” IEICE Technical Report, Vol.107， No.176， pp.77-82， 2007.

[22] T. Nagaoka, K. Okano, and S. Kusumoto: “Abstraction of Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop,” IEICE Technical Report, vol.107, to appear， 2008.