Parallel Multiple Counter-Examples Guided Abstraction Loop
— Applying to Timed Automaton—

Kozo Okano†, Takeshi Nagaoka‡, Toshiaki Tanaka‡, Toshifusa Sekizawa§, and Shinji Kusumoto‡

†Faculty of Engineering, Shinshu University, Japan
‡Graduate School of Information Science and Technology, Osaka University, Japan
§College of Engineering, Nihon University, Japan

okano@cs.shinshu-u.ac.jp,
sekizawa@cs.ce.nihon-u.ac.jp,
kusumoto@ist.osaka-u.ac.jp

Abstract - A model checking technique proves that a given system satisfies given specifications by searching exhaustively a finite transition system which represents the system’s whole behavior. If the system becomes large, it is impossible to explore the whole states in reasonable time due to both of CPU time used and memory space where the model is stored. This is called the state explosion problem. One of the solutions to avoid the state explosion problem is using a model abstraction technique. In usual, constructing such an abstract model from the original model becomes error-prone. Hence, automatic generation techniques of abstract models are studied. Especially, Counter-Example Guided Abstraction Refinement (CEGAR) is considered as a promising technique because it automatically refines abstract model if the result is spurious, starting from a small abstract model. We have already proposed a concrete CEGAR loop for a timed automaton. This iteration loop refines the model in fine granularity level. It avoids the state explosion, however, the number of loops increases. This paper proposes a revised technique where multiple counter-examples are simultaneously applied in the refinement step of CEGAR. This device reduces the number of iteration loops. Experimental results show the improvement.

Keywords: CEGAR, Timed Automaton, Model Checking

1 INTRODUCTION

Recently, information systems play important roles in social activities, thus software reliability becomes important. Model checking techniques [6] prove that a given system satisfies given a specification by searching exhaustively a finite transition system which represents the system’s whole behavior. As systems become larger and more complicated, however, it is difficult to prove the reliability of the systems by model checking, because they need searching for whole states completely. For a large system, it is impossible to explore the whole states in reasonable time. Sometimes its model size becomes larger than physical memory size of typical computers. This is called the state explosion problem.

One of the solutions to avoid the state explosion problem is a model abstraction technique.

In usual, constructing such an abstract model from the original model becomes error-prone. Hence, automatic generation techniques of abstract models are studied.

However, such abstraction techniques in general cannot appropriately control the model size. We want an appropriate abstract model which is small enough to perform model checking and also precise enough to obtain a correct answer by model checking.

In order to obtain a better abstract model, automatic iteration techniques to perform whole cycle of abstraction, model checking, simulation, and refinement, have been studied.

Counter-Example Guided Abstraction Refinement (CEGAR) [8] is the root of such studies.

In verification of real-time systems, a timed automaton is used [4], which can represent behavior of a real-time system. For a timed automaton, a real-valued clock constraint is assigned to each state of finite automaton (called a 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 a traditional way of 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, thus we can perform the model checking on a timed automaton. The size of the model, however, increases exponentially with clock variables; thus, an abstraction technique is needed.

Paper [18] firstly shows a concrete CEGAR loop for timed automata based on predicate abstraction techniques. It uses two abstraction models, over-approximation and under-approximation, while our previous approach [19] constructs an abstraction model based on only over-approximation. Their approaches are similar to our approach in a sense that a location is divided into two state while abstraction. Paper [19] proposed a concrete CEGAR loop for timed automaton. This iteration loop refines the model in fine granularity level. It avoids the state explosion, however, the iteration grows.

1.1 Contributions

This paper proposes a revised technique where multiple counter-examples are simultaneously applied. This device reduces the number of iteration loops.

CEGAR automatically generates a moderate model to perform model checking, but sequential application of counter-examples might consume time and memory space. Our method reduces the number of iteration loops, therefore, it also reduces time and space.

The concrete contributions are summarized as follows.
1. We consider a new CEGAR loop (algorithm) in which multiple counter-examples are simultaneously applied.

2. We give proofs for the algorithm including its termination.

3. We prototyped the algorithm, performed experiments and obtained results which show effective performance of our proposed algorithm.

1.2 Related Work

Other related work includes papers [13, 9, 11, 7, 3, 14], and [16].


None of these approaches, however, deals with refinement with multiple counter-examples.

1.3 Organization of the Paper

This paper is organized as follows. Section 2 presents introductory material related to timed automata. Section 3 presents a short review of our previous proposed CEGAR for timed automata. Section 4 will provide our proposed multiple counter-examples abstraction refinement loop. Section 5 will shortly give explanation on our prototype system. Section 6 and 7 provide experimental results and discussions, respectively. The final section concludes the paper.

2 PRELIMINARIES

Here, we give a definition of a timed automaton and its related notions.

2.1 Timed Automaton

No one can control flow of time. One can only measure time using clocks.

A timed automaton also uses clocks to refer time. The clocks can be regard as precise analog clocks. Every clock autonomously uniformly and at the same rate increases the value, independently from the behavior of timed automaton. A timed automaton cannot control the clocks except for reset; it can neither put some clocks forward, backward nor stop them. It can only reset some of clocks. The reset clocks make their values 0, they, however, immediately increase their values again.

Definition 2.1 (Clock set $C$). By $C$ we denote a finite set of clocks. By $x_i$ ($0 \leq i \leq |C| - 1$) we denote an element (each clock) in $C$.

When there is no confusion we might use literals (without index) $x, y, z$, and so on to denote clocks.

Since each clock has its time value as a non-negative real, notion of “clock evaluation” is needed.

Definition 2.2 (Clock Evaluation). Clock evaluation $\nu(\in \mathbb{R}_{\geq 0}^{|C|})$ for clock set $C$ is a $|C|$-dimension vector over $\mathbb{R}_{\geq 0}$.

An i-th element $\nu_i$ of $\nu$ corresponds to the time value of clock $x_i$.

We use the term “evaluation” according to the original paper [1]. Paper [1] defines the evaluation as a mapping from clocks to reals, however, we define $\nu$ just as a real vector, in this paper. Since clock evaluation changes according to the elapsed time, and a timed automaton might reset some of clocks to 0 when a transition fires, we introduce two operations on clock evaluation.

Definition 2.3 (Operations on Clock Evaluation). For a real value $d, \nu + d = (\nu^0 + d, \nu^1 + d, \ldots, \nu^{|C| - 1} + d)$.

For a set of clocks $r, r(\nu) = (r(\nu^0), r(\nu^1), \ldots, r(\nu^{|C| - 1}))$,

where

\[
\begin{align*}
0 & : x_i \in r, \\
\nu^i & : \text{otherwise .}
\end{align*}
\]

The first operation $+d$ means that every clock increases its value uniformly and at the same rate. The second operation $r(\cdot)$ means that every clock specified in $r$ are reset.

Next we define clock constraints on $C$, which are used as guards and invariants of a timed automaton.

Definition 2.4 (Differential Inequalities on $C$). Syntax of a differential inequality in on a clock set $C$ is given as follows:

\[
in ::= x_i - x_j \sim a$

\[
| x_i \sim a,
\]

where $x_i$ and $x_j \in C$, $a$ is a literal of an integer constant, and $\sim \in \{\leq, \geq, >, <\}$.

Differential inequalities $x_i \sim a$ and $x_i - x_j \sim a$ are true if $\nu^i \sim a$ and $\nu^i - \nu^j \sim a$ are true, respectively.

Definition 2.5 (Clock Constraints on $C$). Syntax of a clock constraint cc on a clock set $C$ is given as follows:

\[
cc ::= \text{true} \mid in \mid cc \land cc,
\]

where $in$ is a differential inequality on $C$.

$cc_i \land cc_j$ is true iff $cc_i$ and $cc_j$ are true.

By $c(C)$, we denote whole set of clock constraints on a clock set $C$.

Since clock constraint $f$ can be regarded as a function $f : C \rightarrow \{\text{true, false}\}$, we introduce a notation $f(\nu)$. It is evaluated to true or false by evaluating each clock $x_i$ as $\nu^i$.

Now we can formulate a timed automaton. The semantics of timed automaton, however, will be defined later through a labelled transition system.
Example 2. Let assume that $C$ and $I(l_2)$ (a location invariant for $l_2$) are $\{x, y\}$ and $y > 6$, respectively. Consider a transition $l_1 \xrightarrow{a,x \geq 0 \wedge y \geq 3}(y) \rightarrow l_2$.

For a clock evaluation $\nu = (8.2, 5.1)$, the values of $r(\nu)$, $g(\nu)$, and $I(l_2)(r(\nu))$ are $(8.2, 0)$, true, and false, respectively.

The following expressions are the deriving processes.

- $r(\nu) = r(8.2, 5.1) = (8.2, 0)$
- $g(\nu) = g(8.2, 5.1) = 8.2 > 0 \wedge 5.1 \geq 3 \Rightarrow true$
- $I(l_2)(r(\nu)) = I(l_2)(8.2, 0) = 0 > 6 \Rightarrow false$

Dynamic of a timed automaton can be expressed via a set of locations and a set of clock evaluations. Changes of one state to a new state can be as a result of firing of an action or elapse of time.

In order to define the semantics of a timed automaton, we firstly define a labelled transition system.

Definition 2.7 (Labelled Transition System). A labelled transition system (LTS) is three-tuple $(S, s_0, T)$, where $S, s_0 \in S$ and $T$ are a finite set of states, an initial state, and a set of transitions, where $T \subseteq S \times (A \cup \mathbb{R}_{\geq 0}) \times S$.

The first and last elements in $S$ stand for states the transition starting from and going to, respectively. $A$ is a finite set of actions.

A transition $(s, a, s')$ of LTS is denoted by $s \xrightarrow{a} s'$.

We can define a run of an LTS.

Definition 2.8 (A Run of an LTS). A run of LTS $(S, s_0, T)$ is defined as follows.

- $s_0 \xrightarrow{a} s'$ is a run of $(S, s_0, T)$, if $s_0 \xrightarrow{a} s' \in T$.

Let $s_i$ be a run of $(S, s_0, T)$, ending with state $s_i$. For $s_i \xrightarrow{a} s_j \in T$, and $s_i, s_i \xrightarrow{a} s_j$ is also a run of $(S, s_0, T)$.

Definition 2.9 (Semantics of a timed automaton). For a given timed automaton $\mathcal{A} = (A, L, l_0, C, I, T)$, its corresponding LTS$(S, s_0, T)$ can be formalized as follows.

Let $S = L \times \mathbb{R}_{\geq 0}^{|C|}$, $s_0 = (l_0, 0)$, where 0 is a $|C|$-dimension vector and each of whose elements is 0.

A transition $s \xrightarrow{a} s'$ is defined by Definition 2.10.

Definition 2.10 (Semantics of transition of a timed automaton). For transition $l_1 \xrightarrow{a \in A} l_2$, its corresponding transition of LTS can be defined as follows.

$$\forall d \leq d' \leq d \iff (l_1, \nu + d') \xrightarrow{a \in A} (l_2, r(\nu) + d)$$

The first one is called an action transition, while the other is called a delay transition.

The first rule can be interpreted as follows. If the current clock evaluation satisfies the guard, and after some of clocks in $r$ are reset, the new evaluation $r(\nu)$ also satisfies the invariant of location $l_2$, then $(l_1, \nu) \xrightarrow{a \in A} (l_2, r(\nu))$ can be fired.

The rest rule can be interpreted as follows. For some real $d$, and any $d'$ such that $d' \leq d$, the obtained clock evaluation $\nu + d'$ satisfies the invariant of location $l_1$, then the control
can stay in location $l_1$, but $d$ units of time has elapsed. In other words, the control can stay in $l_1$ until $d$ units of time has elapsed.

Please note that an action transition does not consume time, while a delay transition consumes time staying in the same location.

**Definition 2.11** (run of a timed automaton). *For a timed automaton $\mathcal{A}$, a run $\sigma$ is a finite or infinite run of its corresponding LTS.*

$$\sigma = (l_0, \nu_0) \xrightarrow{\alpha} (l_1, \nu_1) \xrightarrow{\alpha} (l_2, \nu_2) \xrightarrow{\alpha} \ldots,$$

where $\alpha \in A \cup \mathbb{R}_{\geq 0}$.

In usual, as a run of a timed automaton, we only consider an alternate run of delays and actions, in which delay transitions and action transitions alternately occur.

**Example 3.** One of possible runs of $\mathcal{A}_L$ is

- $(\text{off}, (0)) \xrightarrow{\text{off}, (0.5)} (\text{dim}, (0)) \xrightarrow{\text{off}} (\text{dim}, (9.8)) \xrightarrow{\text{off}} (\text{bright}, (9.8)) \ldots$

Please note that in the run of Example 3, delay transitions and action transitions alternately occur.

For further detail about time automata, refer to [4] and [20].

### 2.2 Model Checking

Model checking of an automaton can be formulated as follows.

**Definition 2.12** (Model Checking).

**Input1:** an automaton $A$

**Input2:** a temporal logic expression $p$

**Output:** $A \models p$ or $A \not\models p$

**Output (optional):** If $A \not\models p$, then a counter-example $CE$

In usual, Computational Tree Logic (CTL) is used as a temporal logic for a timed automaton [4].

Intuitively $A \models p$ means that the behavior (possible runs) of $A$ satisfies the property expressed in $p$. Automaton $A$ is also called a model. Thus, model checking is checking process whether a logic expression $p$ holds under the model represented in $A$.

Typical properties are $\text{AG}q$, $\text{EF}q$ and so on. $\text{AG}q$ and $\text{EF}q$ mean that “for any path, always $q$ holds,” and “for some path, eventually $q$ holds,” respectively. $\text{AG}$ and $\text{EF}$ are called temporal operators.

For a state $s$, we can consider a property $\neg \text{EF}s$, which means that starting from the initial state, the automaton cannot reach the state $s$.

**Definition 2.13** (Reachability Problem). *Model checking of a property $\neg \text{EF}s$ (on $A$) is called reachability problem on $A$."

Reachability problem is a fundamental and essential problem for model checking since the algorithm for reachability problem is core of that of general model checking algorithm.

In this paper, we consider only reachability problem.

Counter-example $CE$ is usually a run of automaton $A$ which specifies concretely that property $p$ does not hold.

For reachability problem, its counter-example is a run to reach state $s$.

Nevertheless the number of states produced by a timed automaton is infinite due to the cardinality of reals, reachability problem is decidable [4], since time space can be divided into finite equivalence classes.

In papers [4] and [10], a data structure DBM is introduced to represent clock constraints. Several operations on DBM are also introduced. Using these operations, we can efficiently calculate time space of timed automata.

**Definition 2.14** (DBM (Difference Bound Matrix)). *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 $C_0 = C \cup \{0\}$. Symbol $0$ is a special variable which means a constant value 0.

The $(i, j)$-th entry $(D_{i,j})$ of the matrix stands for a differential inequality of $x_i - x_j \leq n$, for $x_i, x_j \in C_0$. Suppose there is an inequality $x_i - x_j \leq n$ for $x_i, x_j \in \{<, \leq\}$, the $(i, j)$-th entry $D_{i,j}$ is represented by $(n, \leq)$. 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 $D_{1,i}$, respectively.

A zone is the solution set of a clock constraint that is the maximal set of clock assignments satisfying the constraint [4]. It is well-known that such sets can be efficiently represented and stored in memory as DBMs.

There are several model checkers. Typical model checkers produce one counter-example when a property does not hold.

Algorithms of model checking are essentially exhaustive search of whole possible runs. Therefore, if the number of states becomes larger, the complexity becomes larger exponentially or intractable. Such a situation is called “state explosion.” Thus, we have to reduce the number of states by automatic abstraction.

### 3 CEGAR FOR TIMED AUTOMATA

In usual, CEGAR loop firstly generates small abstract model from the original model. The first abstract model is small enough to perform model checking, however it is usually “over-approximated,” i.e., many states are extremely merged into a same state. Therefore, model checking process usually produces a spurious counter-example for the first abstract model. Using the counter-example, CEGAR loop automatically generates a next abstract model, which has more states than the former. Using the next abstract model, we perform model checking again. Such iteration relaxes the over-approximation step by step. At some point of the iteration, we would obtain an appropriate abstract model for model checking.

#### 3.1 Basic Algorithm

This section provides the base algorithm on abstraction refinement technique for the timed automata given in [18] and [19]. As mentioned above the algorithm in [18] and that of [19] is similar in abstract level. However, this paper proposes an extended method of [19], therefore, we describe the base algorithm based on [19].
The abstraction assumption should hold during CEGAR loop. CEGAR loop [8] consists of the following four steps, namely Initial abstraction, Model checking, Simulation, and Refinement.

Figure 2 shows the basic flow of CEGAR loop.

1. **Initial abstraction**
   - An original model $M_0$ and a property $p$ are given as input, and we abstract the original model $M_0$ and obtain an initial abstract model $M_1$.
   - We abstract the model preserving the abstraction assumption.

2. **Model checking**
   - We perform model checking on the abstract model $M_i$. If a model checker outputs $M_i \models p$, then we can conclude that $M_0 \models p$ by the abstraction assumption. Then, we stop the loop. Otherwise, i.e., the model checker outputs $M_i \not\models p$. Also a counter-example $\hat{\rho}_i$ is generated. We have to check every counter-example in $P_i$ on the original model $M_0$, where $P_i$ is a set of concretized runs on $M_0$, each of which is obtained from $\hat{\rho}_i$ by applying inverse of abstraction function $h$.

3. **Simulation**
   - We check every concretized run in $P_i$ on the original model $M_0$. If one of them is executable on $M_0$, then we conclude that $M_0 \not\models p$, because the found run is a real counter-example on $M_0$ and the property $p$. If none of them is executable on $M_0$, we have to refine $M_i$ so that model checking on $M_{i+1}$ does not produce the counter-example $\hat{\rho}_i$.

   We should notice that checking every run in $P_i$ on $M_0$ can be performed symbolically using symbolical presentation on $P_i$ or $\hat{\rho}_i$. We say that $\hat{\rho}_i$ is spurious when none in $P_i$ is executable on $M_0$.

4. **Refinement**
   - If $\hat{\rho}_i$ is spurious, then we refine $M_i$ so that model checking on $M_{i+1}$ does not produce the counter-example $\hat{\rho}_i$.

The $M_{i+1}$ is obtained automatically using $\hat{\rho}_i$. We repeat the loop by go to Model checking with $M_{i+1}$.

In our previous work [19], we give a concrete algorithm of CEGAR for a timed automaton. In the work, we only consider the reachability property as $p$. Thus, we check that $\neg\text{EFl}_{l_e}$, where $l_e$ is an error location. The error location is a location where we think the control never reach.

The following subsections describe the details of each step.

### 3.2 Initial Abstraction

In initial abstraction, we remove all of clock attributes from the given timed automaton [19].

**Definition 3.2** (Abstraction Function $h$). For a timed automaton $\mathcal{A}$ and its semantic model (LTS) $(S, s_0, \Rightarrow)$, an abstraction function $h : S \rightarrow \tilde{S}$ is defined as follows:

$$h((l, \nu)) = l.$$  

The inverse function $h^{-1} : \tilde{S} \rightarrow 2^S$ of $h$ is also defined as $h^{-1}(\tilde{s}) = (l, D_{(l)})$ where $\tilde{s} = l$ and $D_{(l)}$ is a region satisfying $1(l)$ representing by DBM.

**Definition 3.3** (Abstract Model). An abstract model $\tilde{M} = (\tilde{S}, \tilde{s}_0, \Rightarrow_{\tilde{s}})$ of a given timed automaton $\mathcal{A}$ is defined as follows:

- $\tilde{S} = L$;
- $\tilde{s}_0 = l_0$; and
- $\Rightarrow_{\tilde{s}} = \{(l_1, a, l_2) | l_1 \stackrel{a}{\Rightarrow} l_2 \in T\}$.

For $\mathcal{A}$, using its LTS $(S, s_0, \Rightarrow)$, we can say that $\Rightarrow$ is $\{(h(s_1), a, h(s_2)) | s_1 \stackrel{a}{\Rightarrow} s_1' \Rightarrow s_2' \in \Rightarrow\}$. The $i$-th abstract model $M_i = (\tilde{S}_i, \tilde{s}_0, \Rightarrow_{\tilde{s}})$ is obtained from the $i$-th timed automaton $\mathcal{A}_i = (A_i, L_i, l_i, 0, C_i, l_i, T_i)$ by Definition 3.3.

**Definition 3.4** (Abstract Counter-Example). A counter-example on $M = (\tilde{S}, \tilde{s}_0, \Rightarrow_{\tilde{s}})$ is a run of states of $\tilde{S}$ and labels. An abstract counter-example $\hat{\rho}$ of length $n$ is represented in $\hat{\rho} = \tilde{s}_0 \stackrel{a_1}{\Rightarrow} \tilde{s}_1 \stackrel{a_2}{\Rightarrow} \tilde{s}_2 \Rightarrow \cdots \Rightarrow \tilde{s}_n \Rightarrow \tilde{s}_n$. A set $P$ of runs on $\mathcal{A}$ obtained by concretizing a counter-example $\hat{\rho}$ is also defined as follows using the inverse function $h^{-1}$:

$$P = \{s_0 \stackrel{a_1}{\Rightarrow} s_1 \Rightarrow s_2 \Rightarrow \cdots \Rightarrow s_n \mid \bigwedge_{i=0}^{n-1} (s_i \in h^{-1}(\tilde{s}_i) \land d_i \in R_{\geq 0} \land s_i \stackrel{a_i}{\Rightarrow} s_i') \land s_{i+1} \Rightarrow s_i') \}$$.  

We assume that a counter-example is a finite run [19]. We restrict the property to check as reachability, this assumption is reasonable. For a case of loop structures, see [19].

Example 4 shows an example of Initial Abstraction.

**Example 4.** Figure 3 shows a timed automaton and its abstract model.

The original timed automaton is $A_0 (= M_0)$. Its abstract model $M_0$ is just an automaton without clock constraints.
3.3 Model Checking

Abstract model $M_i$ is a just automaton, therefore, we can use several model checkers at this step. In Paper [19], we use UPPAAL to model check. In our new proposed method, however, we use our original model checker in order to produce multiple counter-examples.

**Example 5.** For an abstract model $M_0$ in Fig. 3, a property $\neg EF C$ does not hold, since clearly we can reach state $C$ from the initial state $A$.

Any appropriate model checker outputs $M_0 \not\models \neg EF C$ and its counter-example $A \rightarrow B \rightarrow C$.

3.4 Simulation

Using the DBM library provided by UPPAAL team, we have developed a simulation program. Let $P_i$ be a set of concretized counter-examples produced by $\hat{\rho}_i$, which is a counter-example of $M_i$.

Instead of checking each element of $P_i$, we use DBM and $\hat{\rho}_i$ to simulate on $A_0$ using symbolic simulation technique.

**Example 6.** Figure 4 shows an example process of Simulation.

Simulation checks whether $A \rightarrow B \rightarrow C$ is possible on the original $A_0$ using symbolic simulation technique. At location $A$, we use a DBM structure representing $x = 0 \land y = 0$, which stands for the initial state. Since $A$ has no invariant, we change the DBM structure to represent $x = y$, which shows that clocks $x$ and $y$ increase their values at the same rate. According to the counter-example, we move to location $B$. At location $B$, we obtain a DBM structure representing $x = y \land x < 1$.

3.5 Refinement

The $(i + 1)$-th abstract model $\hat{M}_{i+1}$ is obtained from a timed automaton $\mathcal{A}_{i+1}$ using the abstraction function $h$. The $(i + 1)$-th timed automaton $\mathcal{A}_{i+1}$ is obtained from the $i$-th timed automaton $\mathcal{A}_i$ and a counter-example $\hat{\rho}_i$.

Paper [19] shows a concrete algorithm for refinement (see Appendix A). We call the algorithm Algorithm 1 (or Refinement). Algorithm 1 has two inputs $\mathcal{A}_i$ and $\hat{\rho}_i$, and outputs $\mathcal{A}_{i+1}$.

Figure 5 summarizes the relation among the models. In usual, Algorithm 1 appends additional locations and transitions to $\mathcal{A}_i$ so that $\hat{M}_{i+1}$ can tell two states which are merged in $\hat{M}_i$ as a result of over-approximation.

**Example 7.** We use the same example in Example 6. Figure 6 depicts the result of applying of Refinement t the original timed automaton.
Applying Algorithm 1, the refinement algorithm, we can obtain the refined timed automaton $M_i$ and its corresponding abstract automaton $M_i$.

We can also see that on $M_1$ we cannot reach the error location $C$.

Paper [5] shows that clock conditions in a form of $x - y < c$ cannot be dealt with. Therefore, we assume that the following assumptions in the paper.

**Assumption 1.**

1. We only check reachability: $\neg \exists!E_{\mathcal{L}}$ for model checking.
2. The target timed automaton is diagonal-free, which means that the timed automaton does not contain clock conditions in a form of $x - y < c[5]$.
3. We assume that a counter example is a finite run.

Hereafter, we assume that Assumption 1 always holds in this paper.

## 4 OUR NEW REVISED CEGAR LOOP

Our revised CEGAR loop differs in Model Checking, Simulation, and Refinement from the previous one.

Here, we describe each of them.

### 4.1 Model Checking

Normally, a model checker produces at most one counter-example. In our algorithm, we use master-worker configuration. Each worker performs model checking and generates a counter-example which we expect to be different to others. We describe how each worker generates a counter-example which we expect to be different to others, in Section 5.

### 4.2 Simulation

If one of counter-examples obtained by workers can be executed on $M_0 = M_0$ symbolically, then we conclude that $\mathcal{A} \not\models \neg \exists!E_{\mathcal{L}}$. Otherwise we perform Refinement using the counter-examples.

### 4.3 Refinement

The master gathers counter-examples from the workers, and performs MultipleRefinement (Algorithm 2) shown in Fig. 7.

Using Algorithm 1, Algorithm 2 in Fig. 7 applies each counter-example $\hat{\rho}$ in a given $S_i$. The result is sequentially reflected in the given timed automaton $M_{i+1}$. In the “for loop body,” if a $\hat{\rho}$ is not executable on the current tentative $M_{i+1}$, then for such a counter-example, Algorithm 1 is not applied. The next counter-example in $S_i$ is chosen and the process is repeated.

### 4.4 The Difference between Our Previous Approach and the New Approach

Here, we describe the difference between our previous approach [19] and the new proposed approach.
model $\hat{M}_k$.

Please note that our new method does not fix its Selection Scheme. In other words, its Selection Scheme dynamically changes in every sequence in the loop.

We summarize the difference between the original CEGAR and the proposed CEGAR in Fig. 9.

The following question arises.

Even we assume that $k < n$ holds, we cannot conclude that $\hat{M}_k$ is the same as one of $\hat{M}_1, \ldots, \hat{M}_n$. The reason is that the set $\{\hat{p}_0, \ldots, \hat{p}_{k-1}\}$ is not subset of $\{\hat{p}_0, \ldots, \hat{p}_{n-1}\}$. Please recall that $\hat{p}_i$ is determined by the fixed Selection Scheme and a current abstract model $\hat{M}_i$, while $\hat{p}_i$ is determined by any uncertain Selection Scheme and the initial abstract model $\hat{M}_0$ (of course in general, $\hat{M}_i$).

Regardless the difference, we have to prove that $\hat{M}_k$ is an adequate abstract model.

In this paper, we don’t prove that $\hat{M}_k$ is one of $\hat{M}_1, \ldots, \hat{M}_n$, because it is not correct in logical.

Instead of it, we prove that $\hat{M}_k$ preserves Abstraction assumption for the reachability problem.

4.5 Proof of the Algorithm

The problem is to ensure that Abstraction assumption is preserved for simultaneous application of multiple counter-examples.

Theorem 1 proves that Abstraction assumption is always preserved nevertheless the order of applying multiple CEs might vary.

Theorem 1. For a given set of counter-examples $S_i$, each of which are generated from model checking on $\mathcal{A}_i$, $M_{i+1}$ obtained by Algorithm 2 (and abstraction function $h$) preserves Abstraction assumption.

Before the proof of Theorem 1, we describe the following propositions.

Proposition 1. Termination of Algorithm 1 [19]

Algorithm 1 terminates for reachability problem.

Proposition 2. Preservation of Abstract Assumption [19]

Algorithm 1 preserves Abstraction assumption.

First we give proof of Theorem 2, which is weaker than Theorem 1.

Theorem 2. For a given set of counter-examples $S_i$, each of which are generated from model checking on $\mathcal{A}_i$, if $M_i$ preserves Abstraction assumption, then $M_{i+1}$ obtained by Algorithm 2 (and abstraction function $h$) also preserves Abstraction assumption.

Please note that $S_i$ is a set of counter-examples generated from $\mathcal{A}_i$ with one application of model checking.

The previous approach obtains each $\mathcal{A}_{i+1}$ by applying $\rho_i$ which is generated from $\mathcal{A}_i$ to $\mathcal{A}_i$.

We have to take care of a counter-example which is in $S_i$ but not in $S$.

Theorem 2 holds nevertheless above difference exists. The proof, therefore, uses divide cases.

The following proposition Lemma 4.1 is used in both proofs of Theorems 2 and 3.

Lemma 4.1. If $\rho_j$ is executable on $\mathcal{A}_{(i+1)(j-1)}$ then $\rho_j$ is also a counter-example on $\hat{M}_{(i+1)(j-1)}$.

A proof of Theorem 2 can be given by induction on the number of application of “for loop body” of Algorithm 2.

Proof. Let $j$ be the number of application of “for loop body” of Algorithm 2.

We denote a tentative timed automaton and its abstract model by $\mathcal{A}_{ij}$ and $\hat{M}_{ij}$, respectively. $\mathcal{A}_{ij}$ stands for a tentative timed automaton obtained from $\mathcal{A}_i$ by $j$ times application of “for loop body.” $\hat{M}_{ij}$ also stand for its corresponding abstract model. Therefore, $\mathcal{A}_i = \mathcal{A}_{(i+1)(0)}$ and $\hat{M}_i = \hat{M}_{(i+1)(0)}$ hold.

We use proof by induction, induction on $j$.

Basis: $\mathcal{A}_{(i+1)(0)} = \mathcal{A}_i$. Thus, $\hat{M}_{(i+1)(0)}$ is also $\hat{M}_i$. Hence, from the precondition of Theorem 2, we can say $\hat{M}_{(i+1)(0)}$ preserves Abstraction assumption.

Inductive Step:

Let us assume that we have already performed $(j - 1)$ times the loop body, and obtained a tentative timed automaton namely, $\mathcal{A}_{(i+1)(j-1)}$.

Let also assume that $\hat{M}_{(i+1)(j-1)}$ preserves Abstraction assumption as an inductive assumption.

Now we consider a counter-example $\rho_j$ in $S_i$.

Case 1: $\rho_j$ is not executable on $\mathcal{A}_{(i+1)(j-1)}$.

In such a case, Algorithm 1 is not applied. Thus $\mathcal{A}_{(i+1)(j)} = \mathcal{A}_{(i+1)(j-1)}$ holds. Hence, $\hat{M}_{(i+1)(j)} = \hat{M}_{(i+1)(j-1)}$ also holds. $\hat{M}_{(i+1)(j)}$ also preserves Abstraction assumption by the inductive assumption.

Case 2: $\rho_j$ is executable on $\mathcal{A}_{(i+1)(j-1)}$.

In such a case, Algorithm 1 can be applied. The condition “$\rho_j$ is executable on $\mathcal{A}_{(i+1)(j-1)}$” implies that “$\rho_j$ is also a counter-example on $\hat{M}_{(i+1)(j-1)}$” by Lemma 4.1. By this fact, Proposition 2 and the inductive assumption, we can conclude that $\hat{M}_{(i+1)(j)}$ also preserves Abstraction assumption.

In any case, $\hat{M}_{(i+1)(j)}$ preserves Abstraction assumption.

Proof by induction, we can say that $\hat{M}_{i+1}$ preserves Abstraction assumption.
Figure 10: Model Correspondence between the Methods

Now a proof of Theorem 1 can be also given by induction on $i$.

**Proof. Basis:**
Since $M_0$ preserves Abstraction assumption [19], we can say $M_0$ preserves Abstraction assumption.

**Inductive Step:**
We assume that $M_i$ preserves Abstraction assumption. By Theorem 2 we can prove that $M_{i+1}$ preserves Abstraction assumption.

**Theorem 3.** Termination of CEGAR loop
CEGAR loop using Algorithm 2 terminates.

**Proof.** By Proposition 1 and the fact that $S_i$ is finite set, Algorithm 2 also terminates.

Next we prove the termination of CEGAR loop.

Let a sequence $\hat{\rho}_0, \hat{\rho}_1, \ldots, \hat{\rho}_d$ which are executable counter-examples selected from $S_i$ using Algorithm 2. The index is selection order in Algorithm 2.

From Lemma 4.1, we can say that there is a corresponding loop sequence where each of the loop is application of $\rho_j (0 \leq j \leq d)$, in a VIRTUAL CEGAR loop (See Fig. 10). VIRTUAL CEGAR is a similar CEGAR to our original CEGAR, but it uses different Selection Schemes for each application of the loop body.

Please note that the corresponding $\hat{\rho}$ is chosen from the set of possible counter-examples of the corresponding timed automaton with a certain Selection Scheme. Thus, each Selection Scheme is not the same but dynamically changed in VIRTUAL CEGAR.

In a similar way to paper [19], we can say the size of states in $M_i$ is also finite. Therefore, CEGAR loop terminates.

Figure 11 shows the difference between VIRTUAL CEGAR and our Original CEGAR.

**Example 8.** Let consider a timed automaton in Fig. 4.5. Due to the clock constraints, neither a transition from $B$ to $C$ nor from $D$ to $E$ is firable.

There are two counter-examples: $A \rightarrow B \rightarrow C \rightarrow E$ and $A \rightarrow B \rightarrow D \rightarrow E$.

**Figure 12: An Example Timed Automaton**

First, let’s consider the case the first counter-example is applied. Algorithm 1 generates a copy $B_1$ from location $B$, and generates a transition from $B_1$ to $D$ as well as a transition from $A$ to $B_1$. Finally it removes transition from $A$ to $B$. Next it applies the second counter-example. It generates a new location $D_1$ and removes a transition from $B_1$ to $D$ (in Fig. 4.5).

Next let’s consider the case the second counter-example is firstly applied. It generates a new location $D_1$ then generates also $B_1$. Finally, it removes a transition from $B_1$ to $D$ (in Fig. 4.5).

The result is the same as Fig. 4.5. The application of the first counter-example does not affect the shape of the timed automaton.

Therefore, this example produces the same refinement. Note that we cannot reach location $E$ from $A$ in Fig. 4.5.

**4.6 Modularity of Our Parallel Execution Scheme**

Our proposed parallel execution algorithm is independent of the original CEGAR loop algorithm, therefore for any correct CEGAR loop algorithm, our parallel execution scheme also works correctly. The above proofs also are performed independently of the original CEGAR loop algorithm because it uses assumptions on correctness of it. In other words, these proofs are performed based on modularity scheme.

**5 PROTOTYPE SYSTEM**

Figure 15 depicts the overview of our prototype system. We use RMI framework on Java for communication between
the master and workers. Each worker performs Model Checking and Simulation for its assigned abstract model.

For efficiency, we introduce a modified algorithm, Algorithm 2' shown in Fig. 16.

The major differences between Algorithms 2 and 2' is that Algorithm 2' does not check the executability. It improves the efficiency. However, it means that Algorithm 2' might perform Refinement using pseudo counter-example information. Such a situation, however, does not occur because Algorithm 1 reconstructs $suc\text{\_}list = \langle (l_0, D_0), (l_1, D_1), \ldots, (l_k, D_k) \rangle$ before it transforms the timed automaton. $suc\text{\_}list$ is a feasible path with regard to the counter-example. Therefore, the counter-example is not executable if and only if $suc\text{\_}list$ is an empty list. If $suc\text{\_}list$ is empty, no transformation is performed. Consequently, Algorithm 2' also works correctly.

In our implementation, each worker performs model checking and simulation in the same cycle. This invent reduces cost of exchange of data among model checking and simulation steps.

The abstract model is the same among workers. Thus, we have to give different parameters to workers in order for each worker to generate different counter-examples. As described after we use two strategies to generate counter-examples: shortest traces and the fastest traces. For both of the shortest traces and the fastest traces, the following parameter is used to generate different counter-examples. There might be many shortest (fastest) counter-examples. Among them, what counter-example is chosen by the worker can be a parameter. In order to select different counter-example, we use worker id and random selection for the selection. Of course, if the number of worker is less than that of shortest counter-examples, then some workers might choose the same counter-example.

6 EXPERIMENTS

6.1 Overview

We have performed experiments using two typical examples. One is Fischer’s mutual exclusion protocol. Several process with the same shape of an automaton share a critical section. Mutual exclusion is established in a protocol using clock variables. Therefore, it is a typical symmetric structure.

Another one is Gear Controller [17]. It is a model consisting of an engine, a gearbox, a human interface, a gear controller, and a clutch. It is a parallel system of hetero six components.

Before applying our tool, we need to obtain a single timed automaton presentation of Fischer’s protocol (and Gear controller) since our proposed method cannot deal with a network of timed automata, which is used in UPPAAL verifier in general.

We performed the experiments under the following environment.

Master
CPU: Intel(R) Core™ 2 Duo
CPU L7700 1.80GHz
MM: 2.00GB
The Number of Iterations: Fischer’s protocol

OS: Ubuntu 10.0.4
Workers (14 cpus)
CPU: Dual Core AMD Opteron™ Processor 2210 HE 1.80GHz
MM: 6.00GB
OS: CentOS 5.4

The purposes (research questions) of the experiments are as follows.

1. How efficiently our proposed method works?
2. Are there any difference between:
   (a) types of model structures?
   (b) types of counter-examples used in CEGAR?

Research question 1 can be observed from how CPU times and the number of iteration are reduced in increasing the number of workers.

Research question 2(a) can be observed by comparing the two examples.

Research question 2(b) is hard to answer. We, however, compare using two strategies, the fastest trace and the shortest trace. The fastest trace uses multiple counter-examples with smallest time delay. The shortest traces use multiple counter-examples with shortest (in number of steps) traces. There are many strategies on producing counter-examples. UPPAAL, however, only supports the above two options. Therefore, we think it is reasonable that we compare the two options.

6.2 Results

As CPU time, we measure the elapsed times for the computation. The results are averages of five trials of the same configurations.

Figures 17 and 18 show the results of the number of iteration. The number of nodes stands for the number of workers.

In both of Fischer’s protocol and Gear Controller, the number of iteration decreases according to the number of workers. The shortest trace for Gear Controller has little effect.

Figures 19 and 20 show the results of the CPU times. The performance is improved according to the number of workers, in Fischer’s protocol while Gear Controller shows worse behaviors. The fastest trace also loses its acceleration but the shortest trace requires more time from four workers.

7 DISCUSSIONS

We can see that the numbers of iteration are improved in both of the cases, while CPU times are not. This observation supports that our proposed method is potentially effective (w.r.t RQ1). Also w.r.t RQ2(a) and RQ2(b), we find there are some differences.

However, we have to consider the reason why CPU time is not improved. Two possibilities are considered on the results.

One is the following hypothesis: Refinement with multiple counter-examples certainly refines parts of the automaton, however, which are not essential parts of the automaton for verification of property $p$. Thus, the refinement increases the size of the automaton, which increases CPU time.

The other one is the following hypothesis: The same counter-examples are generated. If some of workers generate the same counter-examples, then the efficiency becomes worse. Such a phenomenon occurs because the random selections do not guarantee that every counter-example is different to others.

Based on the above observations, we have performed the following additional experiments. For the first hypothesis, we have evaluated the number of states. If it increases according to the number of workers, then we can conclude that unnecessary states are generated.

Second we have also evaluated the ratio of unique counter-examples, which is a good index for the second hypothesis.

7.1 The Number of States

Figures 21 and 22 show the number of states. Fisher’s protocol has gradual increase, while fastest trace of Gear Con-
The Number of Nodes
shortest trace
fastest trace

Figure 20: Execution Times: Gear Controller

The Number of States
shortest trace
fastest trace

Figure 21: The Number of States: Fischer’s protocol

troller has strong increase.

7.2 The Quality of Counter-Examples

Figures 23 and 24 show the ratio of unique counter-examples. If the ratio is equal to 1.0 then it means that every counter-example is different to each other. The shortest traces show that increase of the same counter-examples according to the number of workers.

7.3 A Solution

The results support both of the hypotheses. In order to increase the quality of the counter-examples, priority among the counter-examples is considered. Using the priority, we can control level of the refinement by filtering counter-examples used for refinement. We think, however, that there is no silver bullet, in other words the priority cannot be determined statistically and in advance. As an approximate solution, we adopt threshold on the length of counter-examples. The idea is that we only use shorter counter-examples than threshold by the length of the shortest counter-example. From Fig. 19 and 21, we can observe that the shortest trace option is good. Therefore, it is said that the shorter counter-examples are worth to use.

In order to avoid duplication of counter-examples, we think $k$-shortest path algorithm is worth to try. The algorithm is provided by Eppstein [12] and Jiménez [15].

Since UPPAAL uses more sophisticated data structure than DBM which we use and it also uses partial order reduction technique whereas we don’t use any further improvements.

Therefore, we show the comparison between naïve approach and our approach in order to show the improvements.

We think that the experiments show our approach reduces the number of iteration, which also will improve the size of states of abstraction models. The proposed method works better than naïve CEGAR loop does. It is because the proposed method can deal with larger system than the naïve CEGAR, in some cases. The CPU time is also improved. It implies that the main idea that we simultaneously apply the multiple counter-examples will improve the performance because it reduces the number of iteration. We also have to find further improvements such as detecting redundant counter-examples and reducing applies of counter-examples which do not contribute to refinement.

As a conclusion we can say that the main idea that we simultaneously apply the multiple counter-examples will improve the performance, however, there is some room to improve the performance.

8 CONCLUSION

8.1 Summary

This paper proposed a CEGAR loop for timed automata where multiple counter-examples are simultaneously applied. This device strongly reduces the number of iteration loops. The experiments show the promising results. Also we have obtained a candidate criterion for more effective multiple CEGAR.
Figure 24: The Ratio of the Same Counter-Example : Gear Controller

8.2 Future Work

It is a good idea that if the model becomes too large against to a reasonable CPU time deadline, we reconstruct the model using a subset of the previous set of the counter-examples. Such a scheme can control the size of the abstract model finer.

Another idea of future work will be finding effective criteria for filtering better multiple counter-examples. We also want to try the idea that utilizing modular checking provided in paper [13] and to reconstruct our method based on approach in [18]. Extension of the class of the property is also considered. For example, we want to try to provide CEGAR loop for some subset TCTL [2].

ACKNOWLEDGMENTS

This work is partially being conducted as Grant-in-Aid for Scientific Research S (25220003), C (26330092) and also C (16K00094).

References

Kozo Okano received his BE, ME, and PhD degrees in Information and Computer Sciences from Osaka University in 1990, 1992, and 1995, respectively. From 2002 to 2015, he was an Associate Professor at the Graduate School of Information Science and Technology of Osaka University. In 2002 and 2003, he was a visiting researcher at the Department of Computer Science of the University of Kent in Canterbury, and a visiting lecturer at the School of Computer Science of the University of Birmingham, respectively. Since 2015, he has been an Associate Professor at Department of Computer Science and Engineering, Shinshu University. His current research interests include formal engineering, Shinshu University. His current research interests include formal

Takeshi Nagaoka received the MI, DI degrees from Osaka University in 2007 and 2010, respectively. He currently works for Toshiba Solutions Corporation. His research interests include abstraction techniques in model checking, especially a timed automaton and a probabilistic timed automaton.

Toshiaki Tanaka received the BE, MI degrees from Kobe University in 2009 and from Osaka University in 2011, respectively. He currently works for Sony Corporation. His research interests include parallelization of model checking, especially a timed automaton.

Toshifusa Sekizawa received his MSc degree in physics from Gakushuin University in 1998, and Ph.D. in information science and technology from Osaka University in 2009. He previously worked at Nihon Unisys Ltd., Japan Science and Technology Agency, National Institute of Advanced Industrial Science and Technology, and Osaka Gakuin University. He is currently working at College of Engineering, Nihon University. His research interests include model checking and its applications.

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

---

**Algorithm 1: Refinement Algorithm for a counter-example**

**Inputs** \( \mathcal{A}, \hat{\rho} \)

**Output** \( \mathcal{A}_{i+1} \)

\[ \hat{\rho} = l_0 \cdot a_1 \cdot l_1 \cdot a_2 \cdot l_2 \cdot \ldots \cdot a_n \cdot l_n \cdot l_n(\hat{l}_n = e) \]

\( \text{Algorithm 1} \) uses a counter-example \( \hat{\rho} \), and \( l_k \) is the last location reachable from the initial state. \( \text{Algorithm 1} \) shows the algorithm of Refinement, Algorithm 1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{algorithm1.png}
\caption{Algorithm 1: Refinement Algorithm for a counter-example}
\end{figure}

**Appendix A**

Figure 25 shows the algorithm of Refinement, Algorithm 1. Algorithm 1 uses a counter-example \( \hat{\rho} \) and generates a refined timed automaton. It uses functions, Duplications, Removals, and DuplicateInitialLocation(). Functions Duplications, Removals and DuplicateInitialLocation() are functions to duplicate locations and transitions, to remove unnecessary transactions, and to duplicate the initial location, respectively. For the definitions of these functions, please refer to [19].