On the Generation of Human-oriented Counter-examples using a Test Automaton

Chikyu Yanagisawa†, Shinpei Ogata‡, and Kozo Okano*

†Graduate School of Engineering, Shinshu University, Japan
‡* Faculty of Engineering, Shinshu University, Japan
†15tm535d@shinshu-u.ac.jp
‡ogata@cs.shinshu-u.ac.jp
* okano@cs.shinshu-u.ac.jp

Abstract - To ensure quality in the process of software production, various techniques for software testing have been studied, but many conventional tests are known to take a lot of resources. Therefore, formal methods are attracting attention as a way of improving the quality of software. Model checking is one example of a formal method that inspects logically and exhaustively whether a given property is satisfied or not. A counter-example is a trace information to help with the localization of bugs and can be generated by a model checker when a given property does not hold. However, current model checkers often cannot generate counter-examples expected by users due mainly to their searching algorithms. We propose a method which derives an expected counter-example by combining model checking with a test automaton. The derivation method first creates a test automaton which roughly represents the behavior of the expected counter-example and then performs model checking on a parallel composition of the original automaton with the test automaton. We have applied the proposed method to a case study of water tanks of a chemical plant and confirmed its usefulness.

Keywords: counter-example, model checking, automaton

1 INTRODUCTION

With each coming year, advanced information societies need more reliable software and new techniques to develop such software. Software testing methods to ensure quality in the process of software production have been studied for many years. Conventional testing methods are, however, known to take a lot of resources. Therefore, formal methods are attracting attention as a way of improving the quality of software. A formal method is a method which describes the requirements and design of information systems (software and hardware) using a mathematical based language and provides a mechanism to infer that the system satisfies the requirements of users. This study uses model checking [2], which is one type of formal method.

Model checking inspects logically and exhaustively whether given properties are satisfied or not. It creates a model from source codes or systems and derives logical expressions from the requirements specification and inspection items to be entered into a model checker using a formal language. One application of model checking is "post-verifications" of a system. The procedure of post-verification utilizes models of the system when a fault has occurred [10]. It then determines the cause of the fault using formal methods, in contrast to conventional approaches which carry out cause isolation by log analysis of a particular system of failure.

Modern society depends on post-verifications in many situations because system faults can sometimes give serious impacts on human lives. As a specific example of a fault due to a system malfunction, we can recall the system troubles of a Japanese airline company that occurred as recently as 2016 [20]. When we perform post-verifications, a key element is the use of counter-examples. In general, a counter-example is regarded as "an example that refutes or disproves a hypothesis, proposition, or theorem."

When using counter-examples in post-verification, we treat properties of the system which must be fulfilled as "hypotheses, propositions, or theorems". Thus, a counter-example becomes an example of not satisfying important properties, and we can diagnose how a system fails by tracing it. Research has been conducted on how to generate counter-examples that are easy for humans to understand [8, 9].

The generation of counter-examples, however, may not produce ones which a user expects due to the searching algorithms used in a model checker. This trend is especially noticeable in cases where counter-examples include loop structures. In order to solve the problem, this paper proposes a method for creating test automata to guide the creation of counter-examples for a model represented by time automata [4] which are used in UPPAAL [5], an integrated tool for modeling, validation, and verification of real-time systems. To do this, our proposed method begins by creating a coarse behavior series of counter-examples represented in test automata [16–19]. A parallel composition of the test automata and the original automaton lead to counter-examples of the original model. In addition, we applied our technique to the diagnosis of a chemical plant system example and confirmed the effectiveness of the method.

Section 2 defines the time automata and test automaton used in our approach and Section 3 describes the proposed method for constructing counter-examples using test automaton components. Section 4 presents how we used our method to diagnose a system failure that occurred in the chemical plant systems example of our work. A discussion of the results is given in Section 5 and the summary and future work is given in Section 6.

ISSN1883-4566 ©2017 - Informatics Society and the authors. All rights reserved.

2 PRELIMINARIES

2.1 Model Checking

Model checking [2, 3] of an automaton can be formulated as follows.

Definition 2.1 (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 general, computational tree logic (CTL) is used as a temporal logic for a timed automaton [5].

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

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

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

Figure 1 represents the model checking process.

2.2 Timed Automaton

A timed automaton uses clocks to refer to time. The clocks can be regarded as precise analog clocks. Every clock is autonomous and will increase its value at the same uniform rate, independently from the behavior of the timed automaton. A timed automaton cannot control the behavior of the clocks except for a reset; it can neither put clocks forward, backward nor stop them. It can only reset some of the clocks. The reset clocks make their values 0 and immediately begin increasing their values again.

Definition 2.2 (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 can use literals (without an index) x, y, z, etc. to denote clocks.

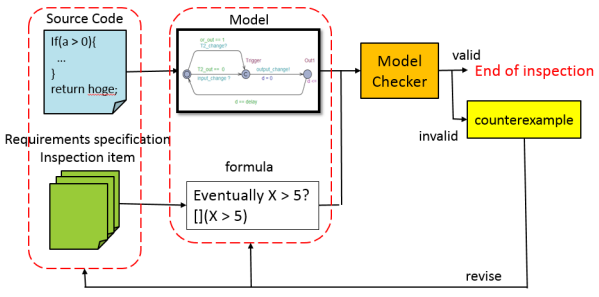

Figure 1: Model checking process.

Each clock has a time value represented as a non-negative real and the notion of "clock evaluation" is needed.

Definition 2.3 (Clock evaluation) *Clock evaluation $\nu(\in \mathbb{R}_{\geq 0}^{|C|})$ for a clock set C is a $|C|$-dimensional vector over $\mathbb{R}_{\geq 0}$.*
The i-th element ν^i of ν corresponds to the time value of clock x_i.

The term "evaluation" is originally defined in [15] as a mapping from clocks to reals. In this work, we define ν simply as a real vector.

In the following definitions we introduce two operations for expressing: 1) clock evaluation value changes according to its elapsed time, and 2) a reset by a timed automaton on some of its clocks when a transition fires.

Definition 2.4 (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 $\mathbf{r}(\subseteq C)$,
$r(\nu) = (r(\nu^0), r(\nu^1), \ldots, r(\nu^{|C|-1}))$, where

$$r(\nu^i) = \begin{cases} 0 : x_i \in \mathbf{r}, \\ \nu^i : \text{otherwise}. \end{cases} \quad (1)$$

The first operation $+d$ means that every clock increases its value uniformly and at the same rate. The second operation $r(\cdot)$ allows us to specify a subset of clocks $\mathbf{r}$ whose values are to be reset to 0.

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

Definition 2.5 (Differential inequalities on C) *The syntax of a differential inequality* **in** *on a clock set C is given as follows:*

$$\mathbf{in} ::= x_i - x_j \sim a$$

$$\mid x_i \sim a,$$

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 *iff $\nu^i \sim a$ and $\nu^i - \nu^j \sim a$ are* true, *respectively.*

Definition 2.6 (Clock constraints on C) *The syntax of a clock constraint cc on a clock set C is given as follows:*

$$cc ::= \text{true} \mid in \mid cc \wedge cc,$$

where in is a differential inequality on C.
$cc_i \wedge cc_j$ is true *iff both cc_i and cc_j are* true.
By $c(C)$, we denote the whole set of clock constraints on a clock set C.

Since a clock constraint f can be regarded as a function

$$f : C \to \{\text{true}, \text{false}\},$$

we introduce the notation of $f(\nu)$ which evaluates to true or false by evaluating each clock x_i as ν^i.

Now we can formulate a timed automaton. The semantics of a timed automaton will be defined later through a labelled transition system.

Definition 2.7 (Timed automaton) *A timed automaton $\mathscr{A}$ is a six-tuple (A, L, l_0, C, I, T), where*
A: a finite set of actions;
L: a finite set of locations;
$l_0 \in L$: an initial location;
C: a clock set;
$I : L \to c(C)$: a mapping from a location to a clock constraint, called a location invariant, or simply an invariant; and
$T \subset L \times A \times c(C) \times 2^C \times L$ is a set of transitions, where $c(C)$ is a set of clock constraints; and 2^C is a super set of sets of clocks.

Elements of the first and last L stand for the locations which the transition is starting from and going to, respectively. An element of A is an action associated with the transition. A clock constraint in $c(C)$ of the transition is called a guard. An element in 2^C is called a set of clocks to be reset.

For conciseness, we denote a transition $(l_1, a, g, r, l_2) \in T$ by $l_1 \overset{a,g,r}{\to} l_2$.

Example 1 *Figure 2 is a depiction of a timed automaton, $\mathscr{A}_L = (\{\text{press}\}, \{\text{off}, \text{dim}, \text{bright}\}, \text{off}, \{x\}, \emptyset, T)$, where* $T = \{\text{off} \overset{\text{press},\text{true},\{x\}}{\to} \text{dim},$
$\text{dim} \overset{\text{press},x\leq 10,\emptyset}{\to} \text{bright},$
$\text{dim} \overset{\text{press},x>10,\emptyset}{\to} \text{off},$
$\text{bright} \overset{\text{press},\text{true},\emptyset}{\to} \text{off}\}$.

Note that guards with value true *and empty clock resets are omitted in Fig. 2.*

Example 1 shows a timed automaton representing the behavior of a mug-light with two brightness modes. Here, we outline the behavior of this time automaton. The initial state is location "off" and the value of clock x is 0. If the "press" action fires, then the state is changed to location "dim", which means that the mag-light is dim. With this transition, the value of clock x is reset to 0. A timed automaton can stay in a location as long as its invariant is satisfied, but this example has no location invariants. The automaton can stay at location "dim" any length of time. If the value of clock x is greater than 10 units of time, the "press" action will change the location to location "off," which means the mug-light is switched off. If the value of clock x is less than or equal to 10 units of time, the "press" action will change the location to location "bright" which means press actions done in succession

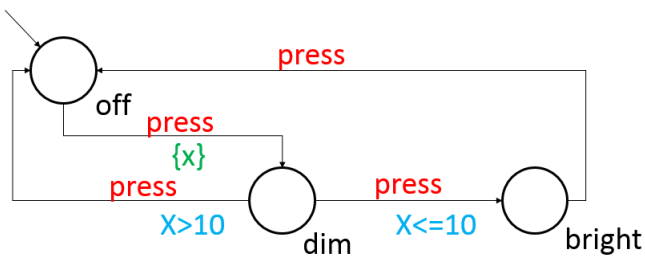

Figure 2: An example of a timed automaton representing a mag-light.

before the clock reaches 10 units of time makes the mug-light bright. At location "bright" the "press" action will change the location to location "off," regardless of the value of clock x.

The following is another example to explain the evaluation of a guard and an invariant.

Example 2 *Let C and $I(l_2)$ (a location invariant for l_2) be $\{x, y\}$ and $y > 6$, respectively.*

Consider a transition $l_1 \overset{a,x>0\wedge y\geq 3,\{y\}}{\to} l_2$.

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, derived as follows:*
$r(\nu) = r(8.2, 5.1) = (8.2, 0)$
$g(\nu) = g(8.2, 5.1) = 8.2 > 0 \wedge 5.1 \geq 3$ =true
$I(l_2)(r(\nu)) = I(l_2)(8.2, 0) = 0 > 6$=false.

The dynamic behavior of a timed automaton can be expressed via a set of locations and a set of clock evaluations. Changes of one state to another can be the result of firing of an action or an elapse of time.

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

Definition 2.8 (Labelled transition system) *For a timed automaton $\mathscr{A}$, a labelled transition system (LTS) is a three-tuple (S, s_0, T), where S is a finite set of states, $s_0 \in S$ is an initial state, and T is a set of transitions, where $T \subset S \times (A \cup \mathbb{R}_{\geq 0}) \times S$.*

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

A transition (s, α, s') of LTS is denoted by $s \overset{\alpha}{\Rightarrow} s'$.
We can define a run of an LTS as follows.

Definition 2.9 (A Run of an LTS) *A run of a LTS (S, s_0, T) is defined as follows.*
$s_0 \overset{\alpha}{\Rightarrow} s'$ is a run of (S, s_0, T), if $s_0 \overset{\alpha}{\Rightarrow} s' \in T$.
Let σ_i be a run of (S, s_0, T), ending with state s_i. Then $s_i \overset{\alpha}{\Rightarrow} s_j \in T$, implies $\sigma_i \overset{\alpha}{\Rightarrow} s_j$ is also a run of (S, s_0, T).

The following define the semantics of a timed automaton.

Definition 2.10 (Semantics of a timed automaton) *For a given timed automaton $\mathscr{A} = (A, L, l_0, C, I, T)$, its corresponding LTS$(S, s_0, T)$ can be formalized as follows.*
$S = L \times \mathbb{R}^{|C|}_{\geq 0}$.
$s_0 = (l_0, \mathbf{0})$, where $\mathbf{0}$ is a $|C|$-dimensional vector each of whose elements is 0.

Definition 2.11 (Semantics of transition of a timed automaton) *For a transition $l_1 \overset{a,g,r}{\to} l_2$, its corresponding transition of LTS can be defined as follows.*

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

The first part is called an action transition, and the second 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 the clocks in r are reset, and the new evaluation $r(\nu)$ also satisfies the invariant of location l_2, then $(l_1, \nu) \stackrel{a}{\Rightarrow} (l_2, r(\nu))$ can be fired.

The second rule is 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 and the control can stay in location l_1, but d units of time will have elapsed. In other words, the control can stay in l_1 until d units of time have elapsed.

Note that an action transition does not consume time, but a delay transition will consume time while staying in the same location.

Definition 2.12 (Run of a timed automaton) *For a timed automaton $\mathscr{A}$, a run σ is a finite or infinite run of its corresponding LTS.*

$\sigma = (l_0, \nu_0) \stackrel{\alpha_1}{\Rightarrow} (l_1, \nu_1) \stackrel{\alpha_2}{\Rightarrow} (l_2, \nu_2) \stackrel{\alpha_3}{\Rightarrow}, \ldots$,
where $\alpha \in A \cup \mathbb{R}_{\geq 0}$.

In general, for a timed automaton, we only consider runs in which delay transitions and action transitions alternately occur.

2.3 Test Automaton

In this paper, the test automaton described in [16–19] is used as guideline for deriving a desired counter-example by tracing transitions of interest in the original model.

Figure 3 represents the general usage of a test automaton. Normally, model checking uses a logical temporal expression to specify a property to check. However, complex specifications might not be able to be expressed using logical temporal expressions. A test automaton, in general, has more expressive power than a logical temporal expression and can represent a variety of complex behavior. A test automaton is typically used in a parallel composition with the original automaton in order to check a complex property. This is a useful technique, however, it is usually difficult to automatically generate a test automaton fitting the designer's need.

The test automaton TA_{id} used in this paper is just a timed automaton. We create a desirable test automaton from test automaton components using several operators in a process described in Section 3.2. This method creates a single test automaton TA_{id} for the original model M with human input and guidance. Then, it automatically generates the desired counter-example by performing model checking on a parallel composition of M and TA_{id}.

3 PROPOSED METHOD

3.1 Motivation Example

We use an example of a chemical plant system (described in more detail in Section 4) as our target for evaluation. Figure 4 shows the counter-example generated by UPPAAL when we perform model checking for the chemical plant system. Note that the model was obtained from a description of the chemical plant system which has some bugs in its design.

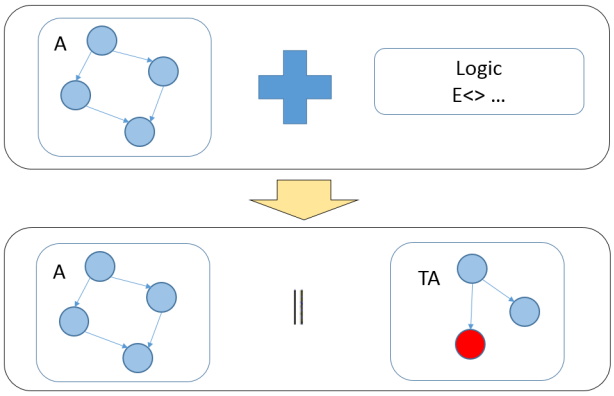

Figure 3: General usage of a test automaton.

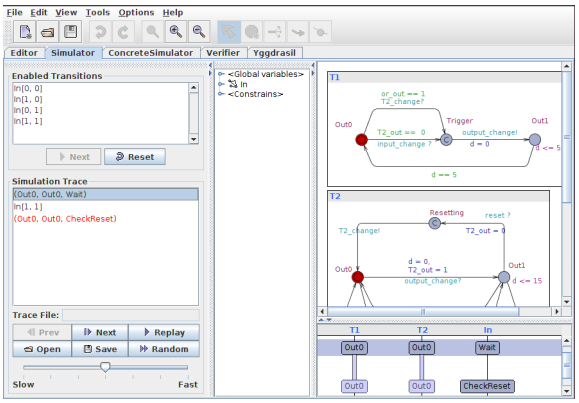

Figure 4: A counter-example generated in model checking of a chemical plant system.

We use the following expression as a property to check.

$$(In.C2 \wedge (T2\_out == 0)) - - > T1.Out1 \qquad (2)$$

This expression uses the "lead to" operation $- - >$ and the expression is equivalent to

$$\mathbb{AG}((In.C2 \wedge (T2\_out == 0)) imply \mathbb{AF}\ T1.Out1) \qquad (3)$$

which means that "when the proposition $In.C2 \wedge (T2\_out == 0)$ is satisfied, the state in which proposition $T1.Out1$ is true will surely be reached at some time."

When this property does not hold, we expect the model checker to generate a counter-example with more detailed information which can be used in subsequent failure diagnosis. In particular, for this example we expect to see as a counter-example the path which cannot reach the state $T1.Out1$ starting from state $In.C2 \wedge (T2\_out == 0)$.

However, the counter-examples actually generated specify only an initial state, which are clearly insufficient for use in fault diagnosis.

In order to solve the above problem, we propose a method using test automata in Section 3.2.

3.2 Method for Generating Test Automata

Our proposed method obtains a useful counter-example by performing model checking on a parallel composition of the original model which represents the behavior of the target system and a test automaton which represents the expected behavior of a counter-example that a user expects to see.

The method begins by creating a rough sketch of a counter-example that users expect as a test automaton. The test automaton is generated by synthesizing "test automaton components". Then, by using a parallel composition of the original model and the synthesized test automaton, we obtain the desired counter-examples in the following steps.

Let M be an original automaton (or automata) to be verified.

1. A user with some domain specific knowledge of the desired system considers an outline of the counter-example to be obtained. The necessary counter-examples are predicted from the system documents and faults report documents.
2. A single test automaton T is composed by synthesizing test automaton components. Weaving of test automaton components is done by using rename and fusion operators.
3. Model checking is performed on a parallel composition of M and T with property P which states "$M||T$ will not reach the final state". If P does not hold, a counter-example is obtained.

In creating the parallel composition, the original model M should be changed as little as possible. However, in order for it to communicate with the test automaton, the following modifications may be required. Note that these modifications can be performed automatically.

If M has a transition with some event $a!$, $a?$, or some variable x which is also used in the test automata, then the following modifications are performed.

1. If M has a transition with event $a!$ $(a?)$, then add a transition which has a synchronization signal $s!$ to a test automaton. Figure 5 shows the modification. Here the location "C" denotes a committed location which is a location where the automaton does not stay. Thus, the event $s!$ is performed immediately after the event $a!$
2. If M has a transition with a variable update $x = expr$ and x is also used in a test automaton, then add a transition which has a synchronization signal $s!$ to the test automaton. Figure 6 shows the modification. Also variable x should be declared as a global variable.

3.3 Test Automaton Components and Operators for Weaving

Figure 7 shows the general form of a test automaton component

$$TA_{id}(L_{in}, a, guard, update, reset, L_{out}, I_s, I_t)$$

Figure 5: Modification of event a!.

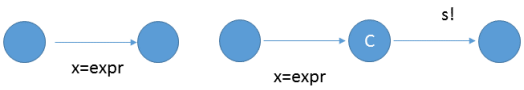

Figure 6: Modification of a transition with update variable.

where L_{in} and L_{out} are the entering location and exiting locations, respectively; $a, guard, update, reset, I_s$, and I_t are an event, a guard, an update, a clock set to reset, invariants for entering and exiting locations, respectively.

In this work, we propose three typical patterns of test automaton components shown in Fig. 8. Let T be the transition of interest in the original model. We consider two types of actions given to the test automaton when tracing the transition of the original model. The first type communicates with transition T and the second communicates with transitions other than T. When transition T fires, the instance of the component of the test automaton simply fires the corresponding transition. This is the first test automaton component. When a other transition other than T fires, we consider the following two scenarios. If the original model can return to a specified state via a transition other than T, i.e., if the original model has a loop, the test automaton must have a transition to the corresponding state. Therefore, a new component is needed. If the original model cannot return to the specified state, i.e., when the original model does not have a loop, the test automaton must have a transition to an error state. If a test automaton reaches an error state, the task of obtaining an appropriate counter-example fails. These three patterns are used as the components of the test automaton.

First we explain the simple component. Figure 8 (i) is a test automaton component to receive signal a and can be written as

$$TA_1(L_{in}, a, true, \emptyset, \emptyset, L_{out}, \emptyset, \emptyset).$$

Using the (i) type test automaton components, we can compose a test automaton which can receive consecutive signals.

For example, let us consider the test automaton components in Fig. 9.

$$TA_1 = (L_{in}, a1?, x > 0, \emptyset, \emptyset, L_{out}, \emptyset, \emptyset)$$

$$TA_2 = (L_{in}, a2?, true, \emptyset, \emptyset, L_{out}, \emptyset, \emptyset)$$

We can use a rename operator ($TA@label\backslash label2$) to change the labels of test automaton components.

For example, $TA_1@L_s\backslash L_{out}$ and $TA_2@L_s\backslash L_{in}$ will rename L_{out} of TA_1 and L_{in} of TA_2 in Fig. 9 each to L_s as shown in Fig. 10.

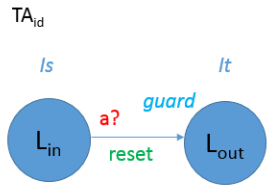

Figure 7: Test automaton component.

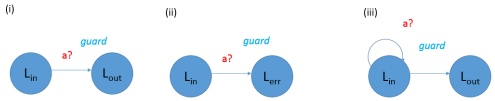

Figure 8: Typical test automaton components: (i) a normal transition, (ii) a force to an error state and (iii) a transition to itself.

The fusion operator $(+)$ is a binary operator which merges locations in both terms.

For example, $(TA_1@L_s\backslash L_{out})+(TA_2@L_s\backslash L_{in})$ will merge L_s of TA_1 and L_s of TA_2 in Fig. 10 to the result shown in Fig. 11.

Figure 8 (ii) forces the test automaton to an error state. It is used when the conditions required for a counter-example are not met.

Figure 8 (iii) stays in a location. It cannot reach location L_{out} while the event happens with a condition guard.

4 CASE STUDY

We have applied the method proposed in Section 3 to the verification of a chemical plant system reported in IPA [10] as a case-study for the "post-verification" of a system.

Figure 12 shows a schematic diagram of the system. The functionality of the system is as follows:

- When the water level is more than a prescribed alert level (40cm), the system opens a discharge valve for 5 seconds to prevent overflow. During this 5-second period and the 10 seconds following it (15 seconds total) the system will not accept a new open instruction.
- The system can also perform the same discharge operation as an instruction from a human operator.
- An instruction from the operator always takes precedence over the other instructions. The system accepts it even during the prohibited interval of 15 seconds. Operator instructions have priority over even past instructions made by the operator.

In operation, even when the water level had exceeded the alert level and the 15-second wait period for new instructions had passed, the appropriate action was not taken by the system. Furthermore, instructions from the human operator which should always have precedence were also ignored. We performed modeling using UPPAAL in order to diagnose this

Figure 9: Test automaton components 1 and 2.

Figure 10: Renamed test automaton components.

failure. Model diagnoses in Figs. 13 and 14 show the control system and a model expressed in UPPAAL, respectively. Figure 14 is derived from the specification of the chemical plant system and the model in Fig. 13.

In Fig. 13, signal $M4$ represents a manual input from the operator and signal $C2$ represents a signal which shows that the water level has exceeded the alert level. For both signals, a 1 indicates an input signal for starting the discharge and a 0 represents no input. Based on these inputs we have two timers for representing the state where the system is draining water and the 15-second wait state when new instructions are not accepted. The specific logic between the inputs and the timers is given in the "In" model (Fig. 14).

In Fig. 14, a double circle represents the initial state. A location with a C represents a committed location. A committed state cannot be delayed and the next transition must involve an outgoing edge of at least one of the committed locations. Expressions in green represent a "guard" which evaluates to a Boolean value. If a guard expression does not hold, the corresponding transition cannot fire. An expression in yellow represents a "select" and in this model, we use it to receive an input from a user (C_2: 0 or 1, M_4: 0 or 1). An expression in blue represents an "update". The model updates variables at the same time as the corresponding transitions. An expression in light blue represents a "synchronization". A synchronization label is formed as either $Expression!$ or $Expression?$. A transition with $Expression?$ fires on receiving the corresponding transition with $Expression!$.

Typically, formal methods decide whether properties which we expect to hold really hold on the model. In this case, it would be natural to consider the negation of a failure that has occurred in the system as a property to check using model checking. In our example, we must determine "whether the system is sometimes put into discharge mode when water draining is not suppressed and the water level sensor is in a state of alert level."

We check Expression (2) in Section 3.1 on the model. As shown in Fig. 4, it is confirmed that this property is not satisfied. The generated counter-example, however, is trivial and it cannot be used to explain why the property is not satisfied. Therefore, it cannot be used for the fault diagnosis of the system.

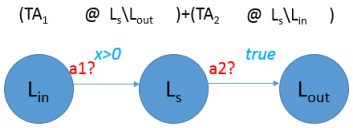

Figure 11: Fusion of test automaton components.

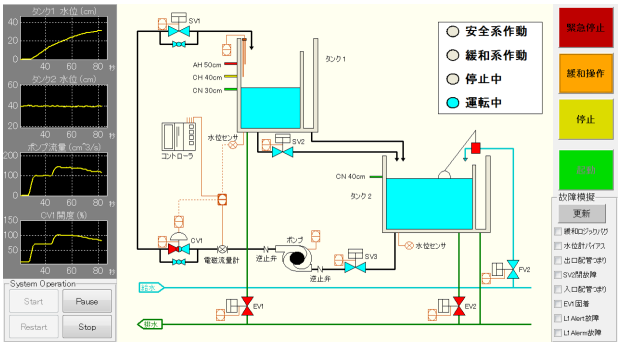

Figure 12: A schematic view of the chemical plant system example from "Fault diagnosis method for the large-scale and complex embedded system (in Japanese)"[10].

Our method creates a test automaton in a series of steps to generate a human expected counter-example. The test automaton is derived by a coarse behavior series of counter-examples in our previous work [10, 11]. Let TA_1 be a test automaton component obtained from type (i) components in Fig. 8.

$$TA_1 = TA_{id}(start, input?, C2 == 1 \&\& M4 == 0,$$
$$isT1end = 1, \emptyset, end, \emptyset, \emptyset)$$

Similarly, TA_2, TA_3, and TA_4 can be obtained from test automaton components in Fig. 8.

Figure 15 summarizes these test automaton components. A test automaton obtained by the following expression is the desired final automaton and is depicted in Fig. 16:

$$(TA_1@L_s\backslash end) + (TA_2@L_s\backslash start, L_t\backslash end)$$
$$+(TA_3@L_s\backslash start) + (TA_4@L_t\backslash start)$$

By combining the model representing the system and the test automaton, a detailed counter-example which can be used in the diagnosis of the failure is obtained(Fig. 17). The odd-numbered lines represent the states of the semantics model of the given network of timed automata. For example, (Out0, Out0, Wait, start) stands for the first timed automaton located in Out0, the second timed automaton located in Out0, and so on. In particular, the last element "start" corresponds to the automaton in Fig. 16. The even-numbered lines represent transitions of the semantics model of the given network of timed automata. For example, "In [1, 0]" on the second line and "In [1, 1]" on the 12th line represents a select input from the user. "input_change: In → T1" on the eighth line represents the fire of a transition "input_change!" on the model "In" and transition "input_change?" fires on the model "T1" also.

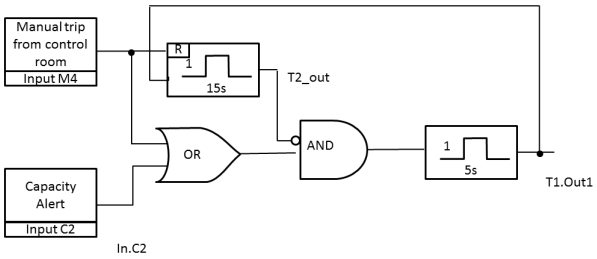

Figure 13: Control diagram of the chemical plant system.

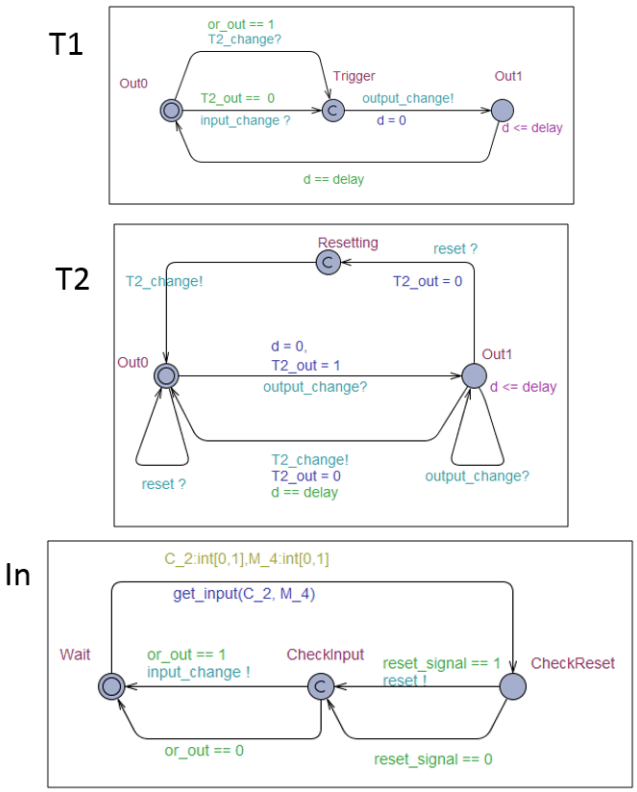

Figure 14: UPPAAL model of chemical plant system example.

5 DISCUSSION

We discuss here the coverage and time taken in generating a counter-example using test automaton components in this work.

The desired counter-example for the case-study presented in this paper was able to be constructed by the proposed test automaton components because the counter-example was generated as a single path. We can expect a variety of cases to be covered sufficiently using the same method.

The time required for counter-example generation was less than 1 second in the example of this work. Since the proposed method uses a test automaton as a guideline for creating counter-examples, it is assumed that the scalability of generation time will fall within an appropriate range even if the counter-example and scale of the time automaton become bigger. In addition, the creation of the test automaton here

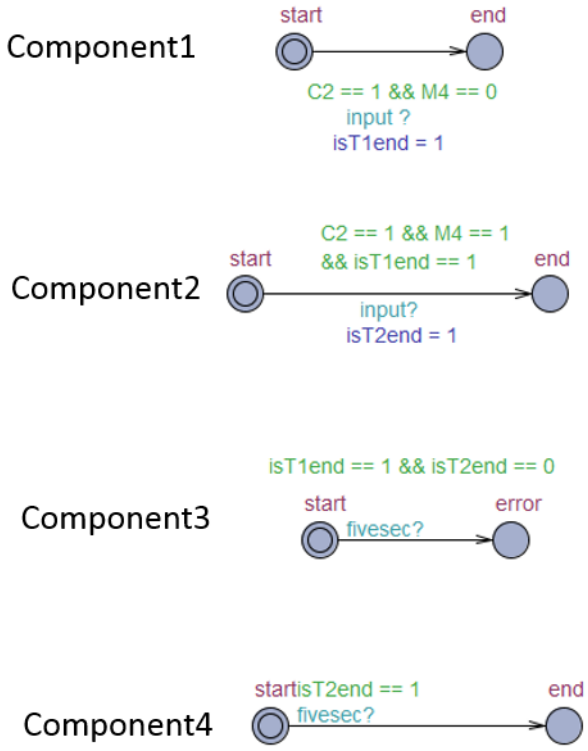

Figure 15: Components of a test automaton for the plant example.

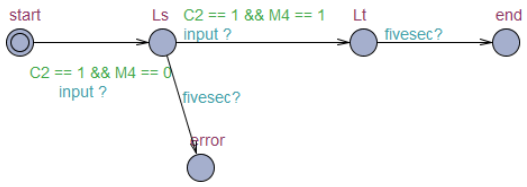

Figure 16: Test automaton.

was by the authors. Therefore, it should be evaluated in the future whether or not a novice in fault diagnosis can create a correct automaton with the prescribed process and how long it would take. This method involves only simple steps of assembling the test automaton from the parts that are provided, however, knowledge of model checking is required to create a useful test automaton. Therefore, the method targets a user with prior knowledge of model checking.

6 CONCLUSION

This paper proposed a method for generating a counter-example which meets user expectations using test automata. We applied the technique to a case study of a chemical plant system in order to verify its effectiveness.

The proposed method is not designed to generate counter-examples automatically. This is an approach that combines model checking and test automata with human guidance to

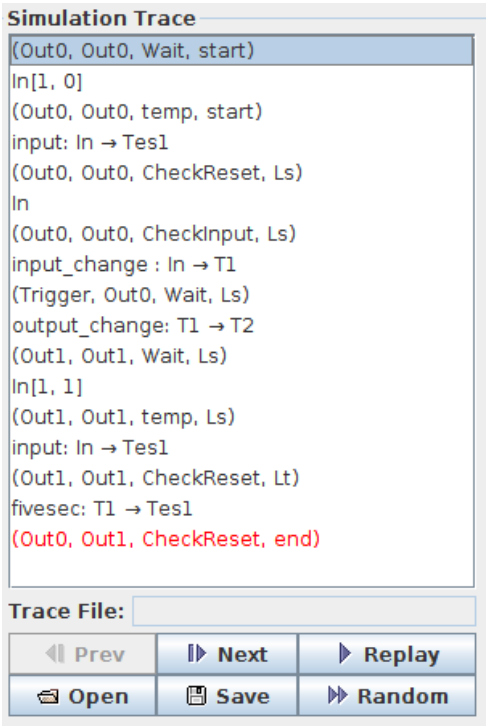

Figure 17: The counter-example generated with the proposed method.

generate a useful counter-example. Therefore, a method for generating counter-examples automatically is needed in the future and we must consider the following problems:

- We cannot fully search the model space in the case of an infinite state transition system.
- Even in a finite state transition system, state-explosion problems can occur.

Therefore, model abstraction techniques to properly reduce the number of states of a model for each property [13, 14] become important. Bounded model checking (BMC) [12] would provide another approach. The BMC technique prevents state explosion by limiting the search range of the finite state space. When BMC finds a violation on a finite state space, counter-examples are generated as finite lengths. In general, counter-examples have infinite length. However, users typically want finite counter-examples and consider a counter-example generated by BMC enough for their purposes. In our future work, we will examine BMC as a promising starting point for generating counter-examples of suitable length within a reasonable amount of time.

ACKNOWLEDGMENTS

This work is being conducted partially as Grant-in-Aid for Scientific Research S (25220003) and C (16K00094). Funding from Mitsubishi Electric Corp. is gratefully acknowledged. Finally the authors wish to acknowledge Dr. Nakajima, Professor of NII for discussion.

REFERENCES

[1] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services Uses Formal Methods," Communications of the ACM, Vol. 58, No. 4, pp. 66–73 (2015).

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, Model Checking, MIT Press, (2000).

[3] K. Okano, T. Nagaoka, T. Tanaka, T. Sekizawa, and S. Kusumoto, "Parallel Multiple Counter-Examples Guided Abstraction Loop —Applying to Timed Automaton—," International Journal of Informatics Society, Vol. 8, No. 3, to appear (2016).

[4] R. Alur, "Timed Automata," Proceedings of 11th International Conference of Computer Aided Verification, (CAV '99), Vol. 1633, pp. 8–22 (1999).

[5] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in Lecture Notes on Concurrency and Petri Nets, Vol. 3098, pp. 87–124 (2004).

[6] G. Behrmann, A. David, and K.G. Larsen, "A Tutorial on Uppaal," Formal Methods for the Design of Real-Time Systems, Vol. 3185, pp. 200–236, (2004).

[7] T. Sekizawa, K. Okano, A. Ogawa, and S. Kusumoto, "Verification of a Control Program for a Line Tracing Robot using UPPAAL Considering General Aspects," International Journal of Informatics Society, Vol. 6, No. 2, pp. 79–87 (2014).

[8] F. Weitl and S. Nakajima, "Incremental Construction of counterexamples in Model Checking Web Documents," Proceedings of the 6th International Workshop on Automated Specification and Verification of Web Systems, EPiC Series, Vol. 18, pp. 61–75 (2010).

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," Journal of the ACM, Vol. 50, No. 5, pp. 752–794 (2003).

[10] IPA, Fault diagnosis method for the large-scale and complex embedded system (in Japanese), https://www.ipa.go.jp/files/000045158.pdf, pp. 68–78 (2015). ⟨accessed June 05, 2016⟩.

[11] K. Okano and J. Kitamiti, Fault diagnosis method for the large-scale and complex embedded system (in Japanese), Software symposium 2015, http://sea.jp/ss2015/paper/ss2015_C1-2(2).pdf, (2015) ⟨accessed July 28, 2016⟩.

[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," Proceedings of 5th International Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS '99), pp. 193–207 (1999).

[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut, "Counterexample-guided Abstraction Refinement," Proceedings of 12th International Conference of Computer Aided Verification (CAV '00), Vol. 1855, pp. 154–169 (2000).

[14] E. Clarke, A, Gupta, J. Kukula, and O. Strichman, "SAT based Abstraction-Refinement using ILP and Machine Learning Techniques," Proceedings of 14th International Conference of Computer Aided Verification (CAV '00), Vol. 2404, pp. 695–709 (2002).

[15] R. Alur, "Techniques for automatic verification of real-time systems," Ph.D. dissertation, Stanford University (1991).

[16] L. Ageto, P. Bouyer, A. Burgueño and K. G. Larsen, "Model-Checking via Reachability Testing for Timed Automata," Proceedings of 4th Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS '98), LNCS Vol. 1384, pp. 263–280 (1998).

[17] L. Ageto, P. Bouyer, A. Burgueño, and K. G. Larsen, "The Power of Reachability Testing for Timed Automata," Journal of the Theoretical Computer Science, Vol. 300, No. 1-3, pp. 411–475 (2003).

[18] B. Bordbar, R. Anane, and K. Okano, "An Evaluation Mechanism for QoS Management in Wireless Systems," International Workshop on Performance Modelling in Wired, Wireless, Mobile Networking and Computing 2005, Proceedings of International Conference on Parallel and Distributed Systems (ICPADS 2005), Vol. 2, pp. 150–154 (2005).

[19] B. Bordbar and K. Okano, "Verification of Timeliness QoS Properties in Multimedia Systems," Proceedings of 5th International Conference on Formal Engineering Methods (ICFEM 2003), LNCS Vol. 2885, pp. 523–540 (2003).

[20] "System trouble grounds JAL ' s domestic flights, affects travelers," The Japan Times News, http://www.japantimes.co.jp/news/2016/04/01/national/system-trouble-grounds-jals-domestic-flights-affects-travelers/, ⟨accessed Feb. 06, 2017⟩.

(Received October 10, 2016)
(Revised January 13, 2017)

Chikyu Yanagisawa received the BE degree from Shinshu University in 2015. He is a master course student of Graduate School of Science and Technology, Shinshu University. His current research interests include formal methods for software engineering.

Shinpei Ogata Shinpei Ogata is an assistant professor of Graduate School of Science and Technology in Shinshu University, Japan. He received a PhD from Shibaura Institute of Technology, Japan in 2012. His current research interests include model-driven engineering for information system development. He is a member of IEEE, ACM, IEICE, IPSJ.

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 the Department of Computer Science and Engineering, Shinshu University. His current research interests include formal methods for software and information system design. He is a member of IEEE, IEICE, IPSJ.