A METHOD TO GENERATE CONFORMANCE TEST SEQUENCES FOR FSM WITH TIMER SYSTEM CALL

Takanori Mori†, Kohei Tokuda†,
Harumasa Tada†, Masahiro Higuchi‡ and Teruo Higashino†
†*Dept. of Info. and Math. Sci., Osaka Univ., Toyonaka, Osaka 560-8531, Japan*
{t-mori, k-tokuda, tada, higashino}@ics.es.osaka-u.ac.jp
‡ *School of Sci. and Eng., Kinki Univ., Higashiosaka, Osaka 577-8502, Japan*
higuchi@ee.kindai.ac.jp

Abstract In this paper, we propose a method to generate conformance test sequences for communication protocols modeled as FSM with timers. The test sequences generated by the proposed method can detect any single fault of timer commands or destination states in the transitions on protocol machines. For each single timer command fault, we give sufficient conditions that a given IUT is (or is not) equivalent to its specification. Based on these sufficient conditions, we give a method for generating test sequences. For each destination state fault, we give a test sequence generation method based on Wp-method. To show the usefulness of this method, we have developed a system for generating test sequences, and applied it to DHCP (Dynamic Host Configuration Protocol). As a result, we have generated the test sequences efficiently.

Keywords: conformance testing, test case generation, embedded system, FSM, timer

1. INTRODUCTION

Recently, complex systems consisting of two or more cooperating components are commonly used. In such complex systems, interactions between components cannot be observed and controlled from their environment, usually[1]. Conformance testing for complex systems are discussed in a lot of papers[2, 3, 4].

Communication protocols which use timer functions provided by operating systems (OS) also can be viewed as such complex systems. In this paper, we discuss conformance testing of such protocols. A system considering in this paper consists of a FSM based protocol machine and timers. A protocol machine is modeled as a deterministic finite state machine (DFSM) which uses timer functions of OS through system

calls. The interactions between a protocol machine and timers cannot be observed and controlled from their environment.

In conformance testing for specific subsystems of complex systems, a given complex system is divided into two components; (i) *Spec* : the subsystem of the test target, and (ii) *Context* : the subsystems in the complex system other than *Spec.* The context is assumed to be correctly implemented. A system consisting of IUT (Implementation Under Test) and Context is called SUT (System Under Test). Such testing is called embedded testing[5] or gray box testing[2].

For embedded testing, we must consider the following features of complex systems; (i) although the IUT does not conform to its specification *Spec*, the behavior of the resulting SUT may conform to *Spec·Context*, and (ii) a single fault of IUT may cause multiple faults of SUT.

Embedded testing has been mainly investigated by assuming that each component of SUT is modeled as DFSM[2, 3, 4]. Although timers can be modeled as DFSM, the numbers of states are usually large. We will consider the above features of timers.

The remainder of this paper is organized as follows. In Section 2, we describe the formal model of FSM with timers. In Sections 3 and 4, we discuss a test sequence generating method. In Section 5, we explain an experiment of applying the proposed method to DHCP.

2. COMMUNICATION PROTOCOLS WITH TIMER SYSTEM CALL

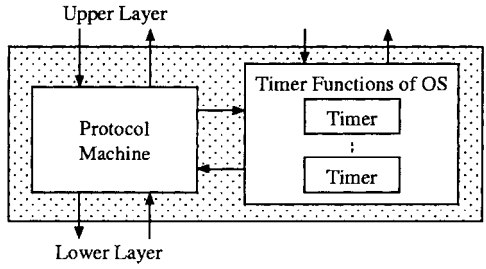

Figure 1. Communication protocols with timer system call

In this paper, we consider a class of communication protocols shown in Figure 1. In this model, a communication entity consists of a finite state protocol machine and multiple timers. The interactions between the protocol machine and timers are (i) timer commands (enabling or disabling timers) from the protocol machine to timers, and (ii) timeout notifications from timers to the protocol machine. Assume that the timer function is provided by operating systems. The timer commands are given as system calls, e.g. `set-timer` or `del_timer` in Linux. The

timeout notifications are given as signals. These interactions cannot be observed and controlled from their environment.

2.1. Timer

In general, operating systems can manage multiple timers. A user process can enable or disable individual timer through system calls. Here, the timer period from enabling a timer to expiring the timer is called the *timer expiring time.* The timer expiring time is specified as a parameter of the system call. After enabling a timer, if the timer expiring time elapses without disabling or re-enabling the timer, then the timer expires and produces a timeout signal.

We assume that each timer can be identified by a timer number and the timer expiring time is fixed for individual timer. A specification of timers is defined by a vector T of timer expiring times. $v[i]$ denotes the i-th element of a vector v. $T[i]$ denotes the timer expiring time of timer i. In order to describe a state of timers, we introduce a timer value vector $\bar{\tau}$. $\tau[i]$ denotes the current value of timer i. $\tau[i]$ has an integer value ($0 \leq \tau[i] \leq T[i]$) or $\perp$. $\tau[i] = \perp$ means timer i is not active. We assume that $\perp > x$ and $\perp - x = \perp$, for all $x \in N$ (N : the set of non-negative integers). When timer i is enabled, $\tau[i]$ is set to $T[i]$. When timer i is disabled, $\tau[i]$ is set to $\perp$. The values of all $\tau[i]$ decrease one by one simultaneously every time unit. When $\tau[i]$ becomes zero, timer i produces a timeout signal and $\tau[i]$ becomes $\perp$. We define $I[i]$ as the set $\{0\ 1, \ldots, T[i], \perp\}$. T denotes the set of timer value vectors $\{\bar{\tau} \mid \tau[i] \in I[i]\}$.

If there exists a timer whose value is zero, a unit time never elapses. If there exist two or more timers whose values are zero, the timers produce timeout signals in the increasing order of their timer numbers. If a timer i whose value is zero is re-enabled or disabled, $\tau[i]$ is set to $T[i]$ or $\perp$, respectively. Such a timer does not produce a timeout signal.

2.2. Protocol machine

A protocol machine is modeled as a Mealy deterministic finite state machine (DFSM) and defined as the following 6-tuple (Q, X, n, Y, H, s_0).

Q	:	a finite set of states.
X	:	a finite set of external input symbols.
n	:	the number of timers.
Y	:	a finite set of external output symbols.
H	:	a finite set of transitions $(u, v, x, y, \bar{p})$.
		$u, v \in Q$: source state, destination state.
		$x \in (X \cup \{1, \ldots, n\})$: input.
		$y \in Y$: external output.
		$\bar{p} \in \langle \mathtt{S}, \mathtt{D}, \mathtt{N} \rangle^n$: timer command vector.
s_0	:	an initial state.

A state transition is executed when a protocol machine receives an input (an external input or a timeout signal). State transitions caused by external inputs and timeout signals are called external input transitions and timeout transitions, respectively. As protocol machines are deterministic, the current state and an input can determine the destination state and outputs (an external output and a timer command vector) uniquely. We assume that state transitions are executed instantaneously. A timer command vector is a vector $\vec{p} = (p[1], p[2], \ldots, p[n])$. $p[i]$ denotes a command for timer i. The command is either `S`, `D` or `N`. `S` is a command enabling a timer. `D` is a command disabling a timer. `N` denotes null operation for a timer.

We assume that protocol machines are reduced. We also assume that each protocol machine has a reliable reset feature. If a protocol machine receives a reset input, the protocol machine is reset to its initial state and disables all timers. If a protocol machine receives an undefined input, the protocol machine is reset and produces an error output. With such error transitions, protocol machines are completely specified.

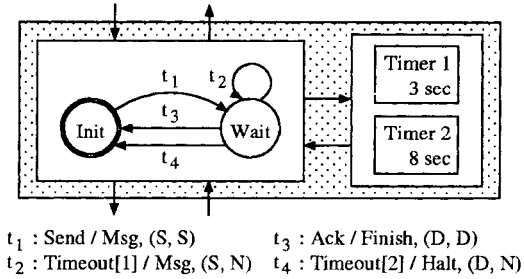

Figure 2. Protocol re-transmitting a message when timeout occurs

Example 1 *Figure 2 is a protocol machine which re-transmits messages when a timer expires. Transmitting a message to a receiver, the protocol machine enables both Timer 1 and Timer 2. The protocol machine re-transmits the message every expiration of Timer 1 until receiving an Ack message. Receiving a timeout signal of Timer 2, the protocol machine stops re-transmitting the message. In Figure 2,* Timeout[1] *and* Timeout[2] *denote timeout signals of Timer 1 and Timer 2, respectively.*

2.3. FSM with timers

For a protocol machine $M = (Q, X, n, Y, H, s_0)$ and a timer specification T, if n equals to the dimension of T, $M \cdot T$ denotes the system consisting of M and T. We use a pair $\xi = \langle s, \vec{\tau} \rangle$ of a state and a timer

value vector to describe an entire state of $M \cdot T$ where $s \in Q$ and $\bar{\tau} \in \mathcal{T}$. Such a pair is called a *composite state* of $M \cdot T$. The set of all composite states of $M \cdot T$ is denoted by Q_T. The initial composite state of $M \cdot T$ is $\xi_0 = \langle s_0, \perp^n \rangle$, where s_0 is the initial state of M, and x^n $(x \in N \cup \perp)$ denotes a vector that all elements of the vector are x.

Given a composite state $\langle s, \bar{\tau} \rangle$, if there exists an integer i $(1 \leq i \leq n)$ such that $\tau[i] = 0$ and $\tau[j] > 0$ for $1 \leq j < i$, then $\langle s, \bar{\tau} \rangle$ is said to be *timer i expiring state*. $\Gamma[i]$ denotes the set of timer i expiring states. $\langle s, * \rangle$ denotes the set of composite states $\{\langle s, \bar{\tau} \rangle \mid s \in Q, \bar{\tau} \in \mathcal{T}\}$. The timer value vector part of a composite state ξ is denoted by $\tau(\xi)$. For the set of composite states Ξ, $\tau(\Xi)$ denotes the set of timer value vectors $\{\tau(\xi) \mid \xi \in \Xi\}$.

We use H_T to denote the set of composite state transitions of $M \cdot T$. H_T is the set of composite state transitions $\eta = (\xi, \xi', x, y)$. There exist three kinds of transitions.

1. external input transitions :
   For an external input transition $(s, s', x, y, \vec{p}) \in H$ there exist composite state transitions $(\langle s, \bar{\tau} \rangle, \langle s', \bar{\tau}' \rangle, x, y)$ on $M \cdot T$ where $\tau[i] > 0 (1 \leq i \leq n)$ and

$$\tau'[i] = \begin{cases} T[i] & (p[i] = \mathtt{S}) \\ \tau[i] & (p[i] = \mathtt{N}) \\ \perp & (p[i] = \mathtt{D}) \end{cases}$$

2. timeout transitions :
   For a timeout transition $(s, s', \mathrm{Timeout}[k], y, \vec{p}) \in H$, there exist composite state transitions $(\langle s, \bar{\tau} \rangle, \langle s', \bar{\tau}' \rangle, -, y)$ on $M \cdot T$ where $\langle s, \bar{\tau} \rangle \in \Gamma[k]$ and

$$\tau'[i] = \begin{cases} T_i & (p[i] = \mathtt{S}) \\ \tau[i] & (i \neq k \wedge p[i] = \mathtt{N}) \\ \perp & (p[i] = \mathtt{D} \vee (i = k \wedge p[i] = \mathtt{N})) \end{cases}$$

3. time elapse transitions :
   There exist composite state transitions $(\langle s, \bar{\tau} \rangle, \langle s, \bar{\tau} - 1^n \rangle, -, -)$ on $M \cdot T$ where $\tau[i] > 0 (1 \leq i \leq n)$.

H_t denotes the set of composite state transitions on $M \cdot T$ corresponding to $t \in H$. Also H_e denotes the set of the time elapse transitions.

For a transition $\eta = (\xi, \xi', x, y)$, $IO(\eta)$ denotes the input/output part of η, i.e. $IO(\eta) = (x/y)$. $\rho(\eta)$ and $\delta(\eta)$ denotes the source composite state and the destination composite state of η, respectively.

A composite state transition string is simply called a string. Given a string $r = \eta_1 \eta_2 \cdots \eta_m \in H_T{}^*$, r is said to be executable from a composite

state $\xi_1 \in Q_T$, if there exist composite states $\xi_2, \ldots, \xi_{m+1} \in Q_T$ such that $\rho(\eta_k) = \xi_k$ and $\delta(\eta_k) = \xi_{k+1}$ for $k(1 \leq k \leq m)$.

Example 2 *A string "$\eta_1\eta_2\eta_3\eta_4\eta_5\eta_6$" is executable from the initial composite state of the protocol machine shown in Figure 2 where $\eta_1 \in H_{t1}$, $\eta_2, \eta_3, \eta_4 \in H_e$, $\eta_5 \in H_{t_2}$ and $\eta_6 \in H_{t_3}$.*

We extend *IO* and δ to accept a string or a set of strings. For a string r, $\delta(r)$ denotes the destination composite state of the last transition of r. $TS(\xi)$ denotes the set of strings which are executable from ξ, and $TS_n(\xi)$ denotes the set of strings in $TS(\xi)$ whose length are n. $R(\xi)$ denotes the set of composite states $\delta(TS(\xi))$. We may explicitly express the target machine M in our notations, like $TS_M(\xi)$ or $R_M(\xi)$.

2.4. Related work

Timed automata[6, 7] are known as a model for describing and analyzing protocols which deal with time dependent behaviors. A timed automaton has several timers. The timer values increase by time elapse, and we can give timing constraints for transitions by simultaneous inequality on timer values. In state transitions, the timer value can be reset to zero. Although we do not give the proof, a timed automaton can simulate our FSM with timers. It means that the procedures for analyzing timed automata are also effective for analyzing FSM with timers.

3. TEST SEQUENCES DETECTING SINGLE FAULT

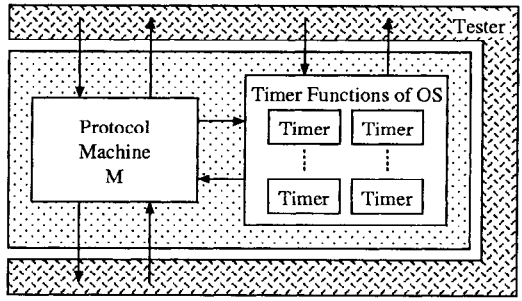

Figure 3. Test architecture

We consider a test architecture for FSM with timers (see Figure 3). In this architecture, the tester can measure time through timer system calls although the tester cannot observe and control the interactions between the protocol machine M and timers. We assume that the tester receives

the timeout signals from timers after receiving all outputs from M. We use this assumption to construct test sequences in Section 3.4.

3.1. Fault model

We introduce a fault model for transitions of the protocol machines as follows.

- timer command faults :
  An element of timer command vector of a state transition differs from that of the specification.
- destination state faults :
  A destination state of a state transition differs from that of the specification.
- external output faults :
  An external output of a state transition differs from that of the specification.

In this paper, we focus to detect single faults. For a given specification $M = (Q, X, n, Y, H, s_0)$ and T, each faulty implementation can be denoted by $M[t'/t] = (Q, X, n, Y, H', s_0)$ and T such that $H' = H \setminus \{t\} \cup \{t'\}$ where t' differs from t only in one element of the timer command vector, the destination state or the external output.

3.2. IO equivalent implementations

In a complex system, although an implementation I does not conform to the specification S, a composed machine $I \cdot C$ (C is a context) may conform to $S \cdot C$ [2, 3]. It is also true in our model of FSM with timers. It means that some implementations which do not conform to the specification cannot be externally distinguished from the specification in context. We introduce the following IO equivalent relation.

Definition 1 *Given two states ξ_1 of A_T and ξ_2 of B_T, ξ_1 and ξ_2 are said to be IO equivalent, written $\xi_1 \equiv \xi_2$, if $IO(TS(\xi_1)) = IO(TS(\xi_2))$.*

Definition 2 *A_T and B_T are said to be IO equivalent, written $A_T \equiv B_T$, if the initial composite states of A_T and B_T are IO equivalent.*

Given a protocol machine M and an implementation $M[t'/t]$ which contains a single timer command fault or destination state fault, $M[t'/t] \cdot T$ may be IO equivalent to $M \cdot T$. On the other hand, if an implementation $M[t'/t]$ which contains a single external output fault, $M[t'/t] \cdot T$ is not IO equivalent to $M \cdot T$ except the case that the faulty transition is not executable, since our tester can observe external outputs.

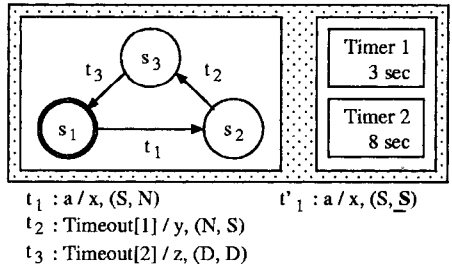

Figure 4. A faulty implementation IO equivalent to the specification

Example 3 *For the protocol machine M shown in Figure 4, let us consider a faulty implementation* $M[t'_1/t_1]$ *in which the transition* t_1 *is implemented as* t'_1.

The initial state of M is $\langle s_1, (\perp, \perp)\rangle$. *In addition, after moving to* s_1 *by transition* t_3, $(\tau_1, \tau_2) = (\perp, \perp)$. *Thus,* $(\tau_1, \tau_2) = (\perp, \perp)$ *holds whenever M is in* s_1. *It also holds for* $M[t'_1/t_1]$.

Let us consider strings which are executable from $\langle s_1, (\perp, \perp)\rangle$ *in M and* $M[t'_1/t_1]$. *After executing* t_1 *or* t'_1, *M is in* $\langle s_2, (3, \perp)\rangle$ *and* $M[t'_1/t_1]$ *is in* $\langle s_2, (3, 8)\rangle$. *For the both composite states, timer 1 will expire after three time elapse transitions. By the timer expiration, transition* t_2 *is executed,* $M \cdot T$ *is in* $\langle s_3, (\perp, 8)\rangle$ *and* $M[t'_1/t_1] \cdot T$ *is also in* $\langle s_3, (\perp, 8)\rangle$. *Furthermore, for the strings corresponding the above behaviors,* $IO(\eta_1\eta_2\eta_3\eta_4\eta_5) = IO(\eta'_1\eta'_2\eta'_3\eta'_4\eta'_5)$ *where* $\eta_1 \in H_{t_1}$, $\eta'_1 \in H_{t'_1}$, $\eta_2, \eta_3, \eta_4, \eta'_2, \eta'_3, \eta'_4 \in H_e$ *and* $\eta_5, \eta'_5 \in H_{t_2}$. *Thus* $M \cdot T \equiv M[t'_1/t_1] \cdot T$.

3.3. Fault detecting sequence

For every implementation $M[t'/t]$, the equivalence between $M \cdot T$ and $M[t'/t] \cdot T$ can be decided by composing two finite automata which accept $IO(TS_{M \cdot T}(\xi_0))$ and $IO(TS_{M[t'/t] \cdot T}(\xi_0))$ and deciding their equivalence. We can compose the above two automata in a similar way to the composition of the region automata from timed automata[6].

Since protocol machines are assumed to be deterministic and completely specified, if $M \cdot T$ is not IO equivalent to $M[t'/t] \cdot T$, there exist at least one IO sequence *io* which is executable from the initial composite state of $M \cdot T$ and is not executable from that of $M[t'/t] \cdot T$. We can adopt such a sequence *io* as a fault detecting sequence.

3.4. Translating IO sequences to test sequences

Let *io* be an IO sequence obtained in the previous section. Since *io* is merely an IO sequence of $M \cdot T$, we have to translate *io* to a test

sequence observable from our tester. Since one or more consecutive timeout transitions occur after at least one time elapse transition, a string of $M \cdot T$ should be an element of the set denoted by a regular expression $\{(E^*F)|(E^+G^+)\}^*E^*$, where E denotes the set of time elapse transitions, F and G denote the set of external input transitions and that of timeout transitions, respectively. The translation is as follows.

- IO sequences corresponding to E^*F :

$$(-/-)^n(x/y) \Longrightarrow \begin{cases} (x/y) & n = 0 \\ (\mathrm{Set}(n)/-)(\mathrm{WE/TO})(x/y) & n > 0 \end{cases}$$

  Set(n) means that the tester sets a timer to n. WE means that the tester Waits Enough until an output comes. TO means a timeout signal corresponding to Set(n). If the tester receives a timeout signal after setting a timer to n, we can guarantee that n time units have passed.

- IO sequences corresponding to E^+G^+ :

$$(-/-)^n(-/-)(-/y_1)\cdots(-/y_m) \Longrightarrow \begin{cases} (\mathrm{Set}(1)/-)(\mathrm{WE}/y_1)\cdots(\mathrm{WE}/y_m)(\mathrm{WE/TO}) & n = 0 \\ (\mathrm{Set}(n)/-)(\mathrm{WE/TO}) & \\ \quad (\mathrm{Set}(1)/-)(\mathrm{WE}/y_1)\cdots(\mathrm{WE}/y_m)(\mathrm{WE/TO}) & n > 0 \end{cases}$$

  We treat E^+ as E^*E. From the assumption, the tester can recognize the timeout signal for Set(1) after observing all outputs $y_1 \cdots y_m$ of timeout transitions.

- IO sequences corresponding to E^* :

$$(-/-)^n \Longrightarrow (\mathrm{Set}(n)/-)(\mathrm{WE/TO}) \quad n > 0$$

Example 4 *For the protocol machine shown in Figure 2, suppose that the transition t_1 is implemented as t'_1.*

$$\begin{aligned} t_1 &= (\mathrm{Init, Wait, Send, Msg,(S, S)}) \\ t'_1 &= (\mathrm{Init, Wait, Send, Msg,(N, S)}) \end{aligned}$$

The following IO sequence is a test sequence detecting the fault.

(Send/Msg)(Set(2)/–)(WE/TO)(Set(1)/–)(WE/Msg)(WE/TO)

The specification accepts the above IO sequence, while the faulty implementation cannot accept it.

4. EFFICIENT TEST SEQUENCE GENERATION

In general, if we enumerate all faulty implementations, check their IO equivalence and generate the corresponding test sequences, the set of the test sequences can distinguish every faulty implementation with a single fault. However, this method may be a obstacle to generating test sequences rapidly. To reduce the time necessary for test sequence generation, we try to deal with multiple faults together and/or generate test sequences by analyzing a smaller FSM than $M \cdot T$.

Since external output faults can be immediately detected by executing the faulty transitions and observing the external outputs, we will focus on detecting timer command faults and destination state faults. We begin with enumerating the reachable states of $M \cdot T$ and generate strings which lead $M \cdot T$ from the initial composite state to each reachable composite state.

4.1. Test sequences for timer command faults

Assume that the specification is given as $M = (Q, X, n, Y, H, s_0)$. An faulty implementation $M[t'/t]$ contains a single timer command fault for timer i on a transition $t = (u, v, x, y, \vec{p}) \in H$.

We will discuss sufficient conditions for $M \cdot T \equiv M[t'/t] \cdot T$ and $M \cdot T \not\equiv M[t'/t] \cdot T$ which can be checked by analyzing smaller machines. To consider such conditions, we introduce the following relation between composite states.

Definition 3 *Given two composite states* ξ *and* ξ', *we write* $\xi \langle i \rangle \xi'$ *if* ξ *and* ξ' *differ only in the value of timer* i.

For $\xi \in Q_T$ and appropriate transitions $\eta \in H_t$ and $\eta' \in H_{t'}$ i.e. $\rho(\eta) = \rho(\eta') = \xi$, $\delta(\eta) \langle i \rangle \delta(\eta')$ or $\delta(\eta) = \delta(\eta')$ holds. Let us consider two composite states ξ and ξ' such that $\xi \langle i \rangle \xi'$, a transition z other than the timeout transition of timer i and two transitions $\eta, \eta' \in H_z$ which are executable at ξ and ξ', respectively. If the timer command for timer i of z is either S or D, $\delta(\eta) = \delta(\eta')$ holds. On the other hand, if the timer command for timer i of z is N, $\delta(\eta) \langle i \rangle \delta(\eta)$ still holds. It means that only strings consisting of the transitions corresponding to transitions whose timer command for timer i is N can distinguish such ξ and ξ'.

By the above consideration, we introduce a sub-protocol machine $M_N = (Q, X, n, Y, H_N, s_0)$ where $H_N = \{z = (s, s', a, b, \vec{p}) \mid z \in H \wedge p[i] = N \wedge a \neq \text{Timeout}[i]\}$.

We introduce the following equivalence between strings.

Definition 4 *Assume that the sets of state transitions of* M *and* $M[t'/t]$ *are* $H = \{t, t_1, \ldots, t_k\}$ *and* $H' = \{t', t_1, \ldots, t_k\}$, *respectively. Two strings* $q (= q_1 \cdots q_m)$ *and* $r (= r_1 \cdots r_n)$ *of* $(TS_{M \cdot T}(\xi_0) \cup TS_{M[t'/t] \cdot T}(\xi 0))$ *are said*

to be $\{t, t'\}$*-equivalent, written* $q \overset{t,t'}{=} r$, *if (1)* $m = n$, *(2)* $q_i \in (H_{t_j} \cup H'_{t_j}) \Leftrightarrow r_i \in (H_{t_j} \cup H'_{t_j})$ *for* $i(1 \leq i \leq m)$, $j(1 \leq j \leq k)$, *(3)* $q_i \in (H_t \cup H'_{t'}) \Leftrightarrow r_i \in (H_t \cup H'_{t'})$ *for* $i(1 \leq i \leq m)$.

We extend the relation "$\overset{t,t'}{=}$" to the relation between the sets of strings.

Definition 5 *Two sets of strings A, B are said to be* $\{t, t'\}$*-equivalent, written* $A \overset{t,t'}{=} B$, *if* $^{\forall}a \in A\ ^{\exists}b \in B\ (a \overset{t,t'}{=} b) \wedge\ ^{\forall}b \in B\ ^{\exists}a \in A\ (b \overset{t,t'}{=} a)$.

Since composite state transitions are translated to IO sequences uniquely and composite state transitions corresponding to t or t' are translated to the same IO sequence, the following lemma holds.

Lemma 1 *For* $M \cdot T$ *and* $M[t'/t] \cdot T$, *if* $TS_{M \cdot T}(\xi_0) \overset{t,t'}{=} TS_{M[t'/t] \cdot T}(\xi_0)$, *then* $M \cdot T \equiv M[t'/t] \cdot T$.

In the following, we will consider a condition to decide $TS_{M \cdot T}(\xi_0) \overset{t,t'}{=} TS_{M[t'/t] \cdot T}(\xi_0)$.

For the strings whose length are one, the following lemma holds.

Lemma 2 *Given two composite states* ξ *of* $M \cdot T$ *and* ξ' *of* $M[t'/t] \cdot T$, $TS_{M \cdot T,1}(\xi) \overset{t,t'}{=} TS_{M[t'/t] \cdot T,1}(\xi')$ *if* $\xi = \xi' \vee \xi <i> \xi' \wedge (\xi, \xi' \notin \Gamma[i])$.

Proof Assume that $\xi <i> \xi' \wedge (\xi, \xi' \notin \Gamma[i])$. Since the executability of external input transitions does not depend on the values of timers, if an external input transition is executable at ξ, then the corresponding transition is also executable at ξ'. Next, since $\xi <i> \xi'$ and timeout transition caused by timer i is not executable at both ξ and ξ', the timeout transition which executable at ξ is $\{t, t'\}$-equivalent to that of ξ'. Both the values of timer i at ξ and ξ' are greater than zero. Hence if the time elapse transition is executable at ξ, then the transition is also executable at ξ'. The case of $\xi = \xi'$ can be shown in a similar way. □

Table 1 describes a necessary and sufficient condition for $TS_{M \cdot T}(\xi_0) \overset{t,t'}{=} TS_{M[t'/t] \cdot T}(\xi_0)$. $\Phi(u)$ denotes the set of timer value vectors at state u i.e. $\tau(\mathcal{R}(\xi_0) \cap \langle u, * \rangle)$. For example, the first row denotes that $M[t'/t]_{\mathrm{N}} \cdot T$ cannot reach $\Gamma[i]$ after executing a transition corresponding to t' at $\langle u, \bar{\tau} \rangle$ $(\bar{\tau} \in \Phi_{M[t'/t] \cdot T}(u) \wedge \tau[i] \neq \perp)$.

Lemma 3 *The followings hold if and only if "condition" holds for every element* $\bar{\tau}$ *in "timer value vector" of the row according to the case of* $p[i]$ *and* $p'[i]$ *in Table 1.*

1. $TS_{M \cdot T}(\xi_0) \overset{t,t'}{=} TS_{M[t'/t] \cdot T}(\xi_0)$.
2. *For each* $r \in TS_{M \cdot T}(\xi_0)$ *and corresponding* $r' \in TS_{M[t'/t] \cdot T}(\xi_0)$, $\delta(r) = \delta(r')$ *or* $\delta(r) <i> \delta(r') \wedge \delta(r), \delta(r') \notin \Gamma_i$.

Table 1. Necessary and sufficient condition for $TS_{M\cdot T}(\xi_0) \overset{t,t'}{=} TS_{M[t'/t]\cdot T}(\xi_0)$

$p[i]$	$p'[i]$	necessary and sufficient condition		
		timer value vector	condition	
S	N	$\bar{\tau} \in \Phi_{M[t'/t]\cdot T}(u)$	$\tau[i] \neq \perp$	$\mathcal{R}_{M[t'/t]_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$ $(\eta \in H_{t'} \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
			$\tau[i] = \perp$	$\mathcal{R}_{M_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$ $(\eta \in H_t \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
N	S	$\bar{\tau} \in \Phi_{M\cdot T}(u)$	$\tau[i] \neq \perp$	$\mathcal{R}_{M_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$ $(\eta \in H_t \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
			$\tau[i] = \perp$	$\mathcal{R}_{M[t'/t]_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$ $(\eta \in H'_{t'} \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
N	D	$\bar{\tau} \in \Phi_{M\cdot T}(u)$	$\tau[i] \neq \perp$	$\mathcal{R}_{M_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$ $(\eta \in H_t \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
D	N	$\bar{\tau} \in \Phi_{M\cdot T}(u)$	$\tau[i] \neq \perp$	$\mathcal{R}_{M[t'/t]_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$ $(\eta \in H'_{t'} \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
D	S	$\bar{\tau} \in \Phi_{M\cdot T}(u)$	$\mathcal{R}_{M[t'/t]_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$	$(\eta \in H'_{t'} \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$
S	D	$\bar{\tau} \in \Phi_{M\cdot T}(u)$	$\mathcal{R}_{M_{\mathrm{N}}\cdot T}(\delta(\eta)) \cap \Gamma[i] = \emptyset$	$(\eta \in H_t \wedge \rho(\eta) = \langle u, \bar{\tau}\rangle)$

The *if part* of Lemma 3 can be shown in induction using Lemma 2. By Lemma 1 and Lemma 3, if $M\cdot T$ and $M[t'/t]\cdot T$ satisfy the conditions shown in Table 1, $M\cdot T \equiv M[t'/t]\cdot T$ holds. The conditions in Table 1 can be checked by exploring a smaller FSM. Unfortunately, we have to compose the automaton for $M[t'/t]\cdot T$ in the case of $p[i]$ =S and $p'[i]$ =N even with Lemma 3.

In the case that the "condition" does not hold, a string r which leads $M\cdot T$ or $M[t'/t]\cdot T$ from $\langle u, \bar{\tau}\rangle \in \mathcal{R}(\xi_0) \cap \langle u, *\rangle$ to a timer i expiring state is obtained. By adding the transfer string from the initial state to $\langle u, \bar{\tau}\rangle$ in front of r, a string r_0 can be obtained. r_0 leads $M\cdot T$ or $M[t'/t]\cdot T$ from ξ_0 to a timer i expiring state.

Next, we will consider a sufficient condition for $M\cdot T \not\equiv M[t'/t]\cdot T$.

Lemma 4 For $M\cdot T$ *and* $M[t'/t]\cdot T$, *if there exist two* $\{t, t'\}$*-equivalent executable strings* r *of* $M\cdot T$ *and* r' *of* $M[t'/t]\cdot T$, *and if two composite states* $\delta(r) = \langle s, \bar{\tau}\rangle$ *and* $\delta(r') = \langle s, \bar{\tau}'\rangle$ *satisfy either one of the following conditions, then* $M\cdot T \not\equiv M[t'/t]\cdot T$ *holds.*

- $\langle s, \bar{\tau}\rangle \in \Gamma[i] \wedge \langle s, \bar{\tau}'\rangle \notin \Gamma[i] \wedge \tau[j] > 0 \ (i < j \leq n)$.
- $\langle s, \bar{\tau}\rangle \notin \Gamma[i] \wedge \langle s, \bar{\tau}'\rangle \in \Gamma[i] \wedge \tau[j] > 0 \ (i < j \leq n)$.

Proof Suppose that one of the above conditions holds. If the value of timer i is zero at $\delta(r)$ or $\delta(r')$, the values of other timers are not zero. Hence, an output caused by timeout of timer i is observed. □

Based on the above lemmas, we can summarize the test sequence generating procedure as follows.

step I According to the types of faults, check the condition in Table 1. If the condition holds, we do not generate a test sequence. Otherwise, we obtain the above mentioned string r_0.

If r_0 and corresponding r'_0 satisfy the condition of Lemma 4, we can adopt r_0 as a fault detecting string. Otherwise, go to step II.

step II By the way described in Section 3.3, we can obtain a fault detecting string.

4.2. Test sequences for destination state faults

Since the specification of protocol machines is assumed to be reduced, we can use existing test sequence generating methods, e.g. Wp-method [8], UIOv-method[9], to obtain strings which distinguish each state. We choose Wp-method for test sequence generation. However, if a string contains timeout transitions, the string may not be executable. Hence, we must decide whether the string is executable.

Given specification of a protocol machine $M = (Q, X, n, Y, H, s_0)$, we consider to generate a test sequence for detecting destination faults on $t = (u, v, x, y, \bar{p}) \in H$. We generate test sequences step by step. In the first step, we generate strings considering only external input transitions.

step 1 Find a state transition sequence which distinguishes v (the destination state of t) from $w_1 \in (Q \setminus \{v\})$ and contains only external input transitions in M. If such a state transition sequence r exists, we can adopt a string $Tr(u) \cdot \eta_t \cdot R$ as a fault detecting string where η_t is a composite state transition corresponding to a target transition t, $Tr(u)$ is a transfer string which leads $M \cdot T$ from the initial composite state to a composite state $\langle u, \bar{\tau} \rangle$ in which η_t is executable, and R is a string corresponding to r.

step 2 Let $w_2 \in (Q \setminus \{v\})$ be a state which cannot be distinguished from v by any string consisting of external input transitions. Find a state transition sequence $t_1 t_2 \cdots t_n \in H^*$ which distinguishes v from w_2 and decide whether there exists a string in $H_e^* H_{t_1} H_e^* H_{t_2} \cdots H_e^* H_{t_n}$ executable in $M \cdot T$. Recall that H_e is the set of time elapse transitions. If such a string exist, we can generate the fault detecting string by adding some appropriate transfer string.

step 3 If any string in $H_e^* H_{t_1} H_e^* H_{t_2} \cdots H_e^* H_{t_n}$ is not executable, there must be unexecutable timeout transitions t_k in the string. So as shown in Figure 5, if timer i is active in some composite state $\langle s_k, \bar{\tau} \rangle$ and there exists a cycle $c_1 \cdots c_m$ consisting of transitions whose commands for timer i is N, then we try to add the cycles in front of the unexecutable transition $t_k = (s_k, s_{k+1}, \text{Timeout}_i, y, \bar{p})$.

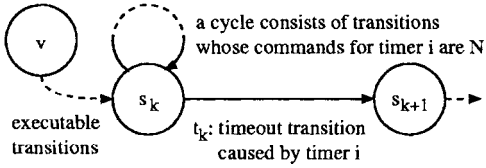

Figure 5. String extension

We also check whether there exists a string in $H_e^* H_{t_1} \cdots H_e^* H_{t_{k-1}} \{H_e^* H_{c_1} \cdots H_e^* H_{c_m}\}^+ H_e^* H_{t_k} \cdots H_e^* H_{t_n}$ executable in $M \cdot T$.

step 4 For a state $w_3 \in (Q \setminus \{v\})$ at which we cannot find a string in the above steps, we can obtain a fault detecting string in the way described in Section 3.3.

In step 1, only a specification M is analyzed. In step 2 and step 3, a specification with timers $M \cdot T$ is analyzed. In step 4, the system consisting of a faulty implementation and timers $M[t'/t] \cdot T$ is analyzed.

5. AN EXPERIMENT

We developed a test sequence generating system based on the proposed method, and applied the system to DHCP (Dynamic Host Configuration Protocol) [10].

5.1. Using simultaneous inequality

In composing a finite automaton of $M \cdot T$, the state space explosion problem arises for protocols containing timers with large $T[i]$ values, e.g. two or more $T[i] = 1000$ timers.

To avoid this problem, we introduce simultaneous inequality to express the set of timer value vectors. The inequalities are classified into the following two types; (i) an inequality $\tau[i] \leq C$ or $\tau[i] \geq C$ specifies the upper limit or the lower limit C (a constant value) of timer i, (ii) an inequality $\tau[i] - \tau[j] \leq C$ specifies the upper limit C of the difference of two timer values $\tau[i]$ and $\tau[j]$. Simultaneous inequality consisting of the above types' inequalities can be solved in $O(lm)$ (l : the number of inequalities, m : the number of variables) [11]. For using such simultaneous inequality, we have made a small modification on our test sequence generating procedure.

A similar method is introduced in verification of timed automata [7].

5.2. A sample protocol

DHCP provides a framework for passing configuration information to hosts on TCP/IP networks. DHCP is built on a client-server model,

where designated DHCP servers allocate network addresses and deliver configuration parameters to clients dynamically. A client requests the use of an address for some period of time (lease time). Both the client and server use timers to watch the lease time. In DHCP, the time unit is a second. The shortest lease time is an hour. A client re-transmits a message based on timeout. The first interval of re-transmitting is four seconds. The interval becomes twice every re-transmission.

If the lease time can be specified by a client, we cannot describe DHCP in our model. Hence, we limit the lease time to an hour, i.e. 3600 time units. In our model, the number of states is 11 and the number of state transitions is *74*. The number of timers is nine where five timers are used to watch the interval of re-transmitting messages. The rest of the timers are used to watch the lease time.

5.3. The result

As a result, we can decide the IO equivalence between the specification and faulty implementation, and generate test sequences for non IO equivalent implementations in a reasonable processing time. The experiment is done on a PC (CPU : Pentium III 600MHz, Memory : 128MB). The computing time and memory space are about seven minutes and 2MB, respectively.

In Table 2, we show the total number of the implementations containing a single fault (test item), the number of IO equivalent implementations, and that of generated test sequences (non IO equivalent implementations). These tables also tell the numbers of faulty implementations shown to be IO equivalent or generated test sequences in every step, which correspond to each step described in Section 4.

Table 2. Result applying our method to DHCP

	timer command
test item	1332
equivalent	457
test sequence	875
step I	1332
step II	0

	destination state
test item	740
equivalent	0
test sequence	740
step 1	705
step 2, 3	35
step 4	0

For every timer command faults, the equivalence was determined and the test sequence was generated for non IO equivalent fault in step I. About 34% of faults are equivalent to $M \cdot T$. Most of these faults are the one that the timer command `N` for non-active timers at source states of target transitions are implemented as `D`.

For the destination state faults, about 95% of the test sequences are generated in step 1, and no test sequence is generated in step 4.

There is no test sequence which is generated by enumerating all reachable states. Test sequences for DHCP are generated efficiently.

6. CONCLUSIONS

In this paper, we have proposed a method to generate conformance test sequences for FSM with timers. The test sequences generated by this method can detect any single fault of timer commands or destination states. We have developed a test sequence generating system and applied it to DHCP. As a result, the test sequences was generated efficiently.

Acknowledgments

This work is partially supported by International Communications Foundation (ICF), Japan.

REFERENCES

[1] B. S. Bosik and M. U. Uyar, "Finite State Machine Based Formal Methods in Protocol Conformance Testing : From Theory to Implementation", Computer Networks and ISDN Systems, Vo1.22, No.1, pp. 7–33 (1991).

[2] L. P. Lima Jr and A. R. Cavalli, "A Pragmatic Approach to Generating Test Sequences for Embedded Systems", In Proc. IFIP 10th International Workshop on Testing of Communicating Systems (IWTCS'97), pp. 288–307 (1997).

[3] A. Petrenko, N. Yevtushenko and G. v. Bochmann, "Fault Models for Testing in Context", In Proc. Joint International Conference on 9th Formal Description Techniques and 16th Protocol Specification, Testing, and Verification (FORTE/PSTV'96), pp. 163–178 (1996).

[4] A. Petrenko and N. Yevtushenko, "Fault Detection in Embedded Components", In Proc. IFIP 10th International Workshop on Testing of Communicating Systems (IWTCS'97), pp. 272–287 (1997).

[5] IS0 9646, "Information Technology, Open System Interconnection, Conformance Testing Methodology and Framework", ISO/IEC 9646 (1991).

[6] R. Alur and D. L. Dill, "A Theory of Timed Automata", Theoretical Computer Science, Vol. 126, No. 2, pp. 183–235 (1994).

[7] R. Alur, "Timed Automata", In Proc. 11th International Conference on Computer-Aided Verification (CAV'99), LNCS 1633, pp. 8–22 (1999).

[8] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, "Test Selection Based on Finite State Models", IEEE Trans. on Soft. Eng., Vol. 17, No. 6, pp. 591–603 (1991).

[9] W. Y. L. Chan, S. T. Vuong and M. R. Ito, "An Improved Protocol Test Generation Procedure Based on UIOs", In Proc. ACM SIGCOMM'89, pp. 283–294 (1989).

[10] R. Droms, "Dynamic Host Configuration Protocol", RFC 2131, Bucknell University (1997).

[11] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to Algorithms", The MIT Press, pp. 539–543 (1990).