A METHOD FOR DESIGNING MULTIMEDIA PROTOCOLS USING BOTH PARAMETRIC MODEL CHECKING AND FUNCTIONAL TESTING

TAKANORI MORI, AKIO NAKATA, AND TERUO HIGASHINO

Abstract. In this paper, we propose a method for designing multimedia protocols using both parametric model checking and functional testing. Especially, we focus on designing media synchronization protocols. We specify a given media synchronization protocol as concurrent periodic timed automata with temporal properties where QoS parameters of the underlying network and timing parameters of the protocol are treated as parameter variables. Based on a parametric model checking method, we automatically derive a condition among parameter variables so that the protocol can execute its transition sequences periodically under the given temporal properties. Next, some parameter values satisfying the derived condition are given. By repeating functional testing for the target IUT, we modify those parameter values so that we can improve the adaptability for the variation of network QoS and/or IUT’s execution environment. We have applied the proposed technique to a simple media synchronization protocol among audio/video streams.

1. Introduction

Recently, according to the progress of the Internet and high speed networks, multimedia communication systems/software, e.g. teleconference and real-time video delivery, are becoming popular. For developing such systems, many multimedia protocols which specify media synchronization among audio/video streams have been proposed [1, 2, 3].

In order to develop adaptable multimedia systems (software) for the variation of network QoS and/or IUT’s (Implementation Under Test) execution environment, we must carefully consider timing aspects for such media synchronization protocols, and decide adequate values for timing parameters of the target protocol such as time necessary for media synchronization, allowable maximum delay of receiving packets and timeout time for thinned-out display of continuous media frames.
Since the values of such timing parameters depend on the QoS parameters in the underlying network, some of them may have to be decided in the implementation level.

In the specification level, both QoS parameters of the underlying network and timing parameters of the protocol are treated as variables. To implement such a real-time protocol, we must decide the values of all parameter values. In this paper, we propose a design method for media synchronization protocols using parametric model checking and functional testing. There exist a lot of research work for parametric model checking and functional testing of real-time protocols [4, 5, 6, 7, 8, 9]. However, to our knowledge, there exist very few work for the design of real-time protocols based on both model checking and functional testing. Our work uses the results of the parametric model checking to avoid deadlocks and unexpected actions. Then, it adjusts given parameter values so that the target IUT behave correctly on the network environments and execution environments by using functional testing.

In the proposed method, we specify a given media synchronization protocol as concurrent periodic timed automata with temporal properties. By using the parametric model checking method proposed in our previous work [4], we automatically derive a predicate consisting of linear inequalities of those parameter variables. The predicate represents the condition that given concurrent timed automata can execute their transitions in the specified time period repeatedly under the given temporal properties. After executing the parametric model checking, we give some standard values satisfying the derived predicate for the parameters. Next, by repeating functional testing for the target IUT, we modify those parameter values. The modified parameter values must also satisfy the derived predicate. By finding adequate parameter values, we can improve the adaptability for the variation of network QoS and/or IUT’s execution environment.

Functional testing tests whether a given IUT has the functions concerning with the designated temporal properties. Although the conformance testing technique may be applied to functional testing of media synchronization protocols, it does not treat temporal aspects in actual execution environments. Ref. [8, 9] have proposed useful methods for functional testing of multimedia systems where timing
aspects of multimedia streams are treated. It treats timing errors and synchronization errors as well as functional errors.

In order to execute functional testing, we decide execution timing for each action in test sequences. Since the outputs from IUTs are uncontrollable and the time necessary for each media synchronization depends on the target IUT’s environment, each output/synchronization timing affects executable time intervals for its succeeding I/O actions. Using the techniques in [14, 15, 16], we decide an executable time interval for each action of a test sequence. When we carry out the proposed functional testing, we can efficiently calculate the executable time intervals for the actions using the techniques.

We have applied the proposed technique to a simple media synchronization protocol among audio/video streams. We explain what predicate can be derived from its specification and temporal properties, and how we carry out the proposed functional testing in order to find adequate parameter values.

The rest of this paper is organized as follows. In Section 2, we define the formal model of concurrent periodic timed automata. We discuss the design and implementation of media synchronization protocols in Section 3. In Section 4, we describe the parametric model checking method. In Section 5, we propose a method to find adequate parameter values using functional testing. Finally, we conclude this paper in Section 6.

2. System Model

For modeling real-time protocols, timed automata are commonly used. Here, we use concurrent and periodic timed automata. In this section, we formally define those concurrent periodic timed automata, and introduce an example media synchronization protocol.

2.1. Timed Automata

A timed automaton [17] is defined as the following 6-tuple \((S, C, P, A, E, s_0)\).

- \(S\) : A finite set of states.
• **C**: A finite set of clocks. The initial values of clocks are zero. Each clock holds a real number.

• **P**: A finite set of parameters. \( \{p_1, p_2, \ldots, p_k\} \).

• **A**: A finite set of actions.

• **E**: A finite set of transitions. \( S \times \bar{A} \times \Phi(C, P) \times 2^C \times S \).

  - \( \bar{A} \) = \( A \times \{?, !\} \). For each \( a \in A \), \( a? \) and \( a! \) denote an input action and output action, respectively.

  - \( \Phi(C, P) \) is a transition condition. Each \( \phi \in \Phi(C, P) \) is defined as \( \phi := c \sim f(P) \mid \phi \land \phi \mid \text{true} \) where \( c \in C, \sim \in \{<, \leq, =, \geq, >\} \), and \( f(P) \) is a linear expression \( a_1p_1 + a_2p_2 + \cdots + a_kp_k + a_{k+1} \) where \( a_i (1 \leq i \leq k+1) \) is a real number and \( p_i \in P (1 \leq i \leq k) \). Note that only the conjunction of linear inequalities is permitted as the transition condition.

  - \( 2^C \) denotes the power set of \( C \).

• **s_0**: The initial state.

Here, \((s, a, \phi, r, s') \in E\) may be written as \( s \xrightarrow{a,\phi,r} s' \). If timed automaton \( M \) is in state \( s \) and transition condition \( \phi \) is satisfied, then \( M \) moves to state \( s' \) by the execution of action \( a \), and all the clocks in \( r \) are reset to zero.

The semantics of timed automata is described as a LTS (Labeled Transition System). We introduce a value assignment \( \sigma \). A value assignment \( \sigma \) assigns a non-negative real number to each variable in \( C \). \((\sigma + d)\) denotes a value assignment where the value of \( \sigma + d \) is assigned to each variable. \((\sigma[r \rightarrow 0])\) denotes a value assignment where if a variable belongs to \( r \), then the value of the variable is zero, otherwise \( \sigma \).

A LTS is defined as a 4-tuple \((Q, q_0, L, \rightarrow)\), where \( Q \) is a finite set of states, \( q_0 \) is the initial state, \( L \) is a finite set of labels and \( \rightarrow \) is a finite set of transitions \((Q \times L \times Q)\). For each \((q_1, l, q_2) \in \rightarrow\), we write it as \( q_1 \xrightarrow{l} q_2 \).

For timed automaton \( M = (S, C, P, A, E, s_0) \), we define LTS \( M_L = (Q, q_0, L, \rightarrow) \) as follows. Each element of \( Q \) is a pair \((s, \sigma)\), where \( s \in S \) and \( \sigma \) is a value assignment. A state \((s, \sigma)\) is called a concrete state. There exist two types of transitions called time elapse transitions and action transitions.
• A time elapse transition \( d \) represents a time passage:

\[
(s, \sigma) \xrightarrow{d} (s, (\sigma + d))
\]

• An action transition \( a \) represents an execution of transition \( a \) in \( E \) if

\[
(s, a, \phi, r, s') \in E \text{ and } \sigma \text{ satisfies } \phi:
\]

\[
(s, \sigma) \xrightarrow{a} (s', (\sigma[r \rightarrow 0]))
\]

### 2.2. Concurrent Periodic Timed Automata

In general, audio/video streams are periodically displayed. Therefore, many media synchronization protocols are executed periodically. In this paper, we model such protocols as concurrent periodic timed automata.

At first, we define periodic timed automata. A *periodic timed automaton* is a timed automaton with an additional time constraint. It must return to the initial state every \( T \) time units. Here, \( T \) is called a *period*. The syntax and semantics of periodic timed automata are the same as non-periodic timed automata.

We can construct a periodic timed automaton \( M' \) from any non-periodic automaton \( M' \) and period \( T \) as follows. We assume that \( M = (S, C, P, A, E, s_0) \) and \( M' = (S', C', P', A', E', s'_0) \) where \( M \) and \( M' \) have the same set of states and the same initial state, i.e. \( S = S' \) and \( s_0 = s'_0 \). We also assume that 

\[
C = C' \cup \{c\}, \quad P = P' \cup \{T\}, \quad A = A' \cup \{\text{reset}\} \quad \text{and} \quad E = E' \cup E_r
\]

where 

\[
E_r = \{(s, \text{reset}, \phi, r, s) \mid s \in S, \phi = (c == T), r = \{C\}\}
\]

Transitions in \( E_r \) mean that if the period \( T \) is elapsed, then automaton \( M' \) returns to the initial state and reset all clocks to zero. Hereafter, we call transitions caused by action \( \text{reset} \) as a *reset transition*.

*Concurrent periodic timed automata* are defined as a set of periodic timed automata. All timed automata in the set must have the same period \( T \) and run concurrently. For all timed automata in the set, all the clocks are running in the same rate. The transitions with the same action name are executed synchronously, and the other transitions are executed asynchronously. Transitions \( t = (s_1, a, \phi, r, s_2) \) in timed automaton \( M \) and \( t' = (s'_1, a, \phi', r', s'_2) \) in timed automaton \( M' \) are executed synchronously if and only if the both transition conditions \( \phi \) and \( \phi' \) hold.
When the same input actions are executed synchronously, a given input is received by automata which execute the input actions. On the other hand, when the same output actions are executed synchronously, automata which execute the output actions produce an output. When input actions and output actions are executed synchronously, the output produced by the latter automata is received by the automata which execute the input actions.

The semantics of concurrent periodic timed automata $\mathcal{M}$ is also defined as LTS $\mathcal{M}_L = (Q, q_0, L, \rightarrow)$. Here, $\mathcal{M} = \{M_1, M_2, \ldots, M_k\}$, and $M_i = (S_i, C_i, P_i, A_i, E_i, s_{0,i})$ (1 $\leq$ $i$ $\leq$ $k$). Each element of $Q$ is a pair of sets $((s_1, \ldots, s_k), (\sigma_1, \ldots, \sigma_k))$, where $s_i \in S_i$ and $\sigma_i$ is a value assignment for $M_i$ (1 $\leq$ $i$ $\leq$ $k$). There exist three types of transitions called synchronized time elapse transitions, synchronized action transitions and asynchronous action transitions.

- A synchronized time elapse transition.
  For any $i \in \{1, \ldots, k\}$, if $(s_i, \sigma_i) \xrightarrow{d} (s_i, (\sigma_i + d))$ on $M_{iL}$, then $((s_1, \ldots, s_k), (\sigma_1, \ldots, \sigma_k)) \xrightarrow{d} ((s_1, \ldots, s_k), ((\sigma_1 + d), \ldots, (\sigma_k + d)))$ on $\mathcal{M}_L$.

- A synchronized action transition.
  For any transition $a$, let $I$ be the set of indices of timed automata such that the transition $(s_i, \sigma_i) \xrightarrow{a} (s_i', \sigma_i')$ can be executed.
  $((s_1, \ldots, s_k), (\sigma_1, \ldots, \sigma_k)) \xrightarrow{a} ((s_1', \ldots, s_k'), (\sigma_1', \ldots, \sigma_k'))$, where $s_j' = s_j$ and $\sigma_j' = \sigma_j$ for $j \notin I$.

  In the case that transition $a$ is a self-loop transition, i.e. $s_i = s_i'$, we can define a synchronized action transition in the similar way.

- An asynchronous action transition.
  If the number of elements in $I$ described above is one, then the transition corresponds to an asynchronous action transition.

2.3. Example Media Synchronization Protocol

Fig. 2.1 is a block diagram of an example media synchronization protocol. The protocol consists of five components: audio buffer module, audio manager, video
A Method for Designing Multimedia Protocols using Both Parametric Model Checking and Functional Testing

Figure 2.1: A block diagram of media synchronization protocol

buffer module, video manager and synchronizer. The arrows in the figure represent flows of audio/video streams. The protocol receives an audio stream and a video stream through the underlying network and plays those streams synchronously.

The audio buffer module receives an audio stream through the network, and analyzes the stream. After that, the audio buffer module informs the arrival of the stream to the audio manager. If an audio stream cannot be received or analyzed, then the audio buffer module ignores the stream. The audio manager plays an audio stream received from the audio buffer module. If an audio stream cannot be received, then the audio manager plays the previous stream again. When the audio manager plays a stream, it synchronizes with the synchronizer. The behavior of the video buffer module and video manager is similar to those of the audio buffer module and audio manager. The synchronizer controls the synchronization timing between the audio manager (audio streams) and the video manager (video streams).

Although we assume one-to-one synchronization between audio and video frames for simplicity of discussion, we can easily extend it to \( m \)-to-\( n \) synchronization where every \( m \)-th audio frame synchronizes with every \( n \)-th video frame repeatedly. In addition, since our media synchronization protocol has a simple synchronizer, we can add audio buffer modules, audio managers, video buffer modules...

SIU 2004
and video managers easily. By adding these modules, we can describe a media synchronization protocol which deals with multiple audio and video streams.

The protocol is specified as the concurrent periodic timed automata in Fig. 2.2. Each transition has a label consisting of an input or output action name, transition conditions (true or linear inequalities) and a set of clocks which are reset by executing the transition (shown as {...}). Although the reset transitions reset described in Section 2.2 are not shown in Fig. 2.2, we assume that each state has the reset transition.

The audio buffer module has a clock tab1 and it is used for controlling an execution timing of each transition. The audio buffer module waits an audio stream until pb1 time units after the previous stream operation has been finished. If the stream arrives at the audio buffer module, then it starts to receive the stream (a_avail?). If pb1 time units have elapsed, the audio buffer module decides that a new stream does not arrive (a_not_avail!) and informs that to the audio manager. Then the
audio buffer module waits for the audio manager to play the previous stream. Action \texttt{a\_not\_avail} is a synchronized action between audio buffer module and audio manager. When the audio buffer module has finished to receive the stream, it informs its arrival to the audio manager (\texttt{a\_wait!}). We assume that it takes at least \( pb_2 \) and at most \( pb_3 \) time units to receive a stream. After receiving the stream, the audio buffer module analyzes the stream. If the stream is correct, then the audio buffer module passes it to the audio manager (\texttt{a\_ready!}). Otherwise, the audio buffer module informs that the stream is wrong to the audio manager (\texttt{a\_late!}). We also assume that it takes at most \( pb_4 \) time units to analyze a stream. The audio buffer module waits for the audio manager to play a stream in both cases. After that, the audio buffer module synchronizes with other modules and returns to its initial state (\texttt{return!}).

The audio manager has two clocks \texttt{tam}_1 \texttt{and} \texttt{tam}_2. Clock \texttt{tam}_1 is used for controlling an execution timing of each transition. Clock \texttt{tam}_2 is used for controlling the period, which is referred in \texttt{return!} transition. The audio manager waits for a message of the stream arrival from the audio buffer module (\texttt{a\_wait?}) until \( pm_1 \) time units. In addition, after \( pm_2 \) time units have elapsed, the audio manager waits for a message informing that a new stream does not arrive (\texttt{a\_not\_avail?}) as well as the stream arrival message. If the audio manager receives the message \texttt{a\_not\_avail}, then it plays the previous stream again (\texttt{play!} which is the transition from state 23 to state 24). On the other hand, if the audio manager receives the message \texttt{a\_wait?}, then it waits for the next message. After that, the audio manager receives the stream from the audio buffer module (\texttt{a\_ready?}) or receives a message which informs that the stream is wrong (\texttt{a\_late?}). If the audio manager receives the stream, then it plays the stream (\texttt{play!} which is the transition from state 22 to state 24). It takes at least \( pm_3 \) and at most \( pm_4 \) time units to decode the stream. In case that the audio manager receives the message \texttt{a\_late?}, it plays the previous stream again (\texttt{play!} which is the transition from state 23 to state 24). It takes at least \( pm_5 \) and at most \( pm_6 \) time units. Action \texttt{play} is a synchronized action among the audio manager, the video manager and the synchronizer. It takes at least \( pm_7 \) time units to play a stream (\texttt{a\_play\_end}). After playing the stream,
the audio manager waits for elapsing the period $T$ and returns to its initial state (return!).

The audio buffer module waits for the arrival of a stream until $pb_1$ time units, and informs the arrival between $pb_2$ and $pb_3$ time units after receiving the stream. On the other hand, the audio manager waits for an arrival message from the audio buffer module until $pm_1$ time units, and $a\_not\_avail?$ can be executed after $pm_2$ time units have elapsed. If parameters $pb_1, pb_2, pb_3, pm_1$ and $pm_2$ do not have adequate values, then the audio buffer module waits for the audio manager to execute action $a\_wait?$ and the audio manager waits for the audio buffer to execute action $a\_not\_avail!$. For example, we assume that $pb_1 = 5$, $pb_2 = 2$, $pb_3 = 4$, $pm_1 = 5$ and $pm_2 = 6$. We also assume that the audio buffer module executes $a\_avail?$ at time four. In this case, $a\_wait!$ can be executed between time six and eight in the audio buffer module. However, $a\_wait?$ can be executed before time five in the audio manager. As a result, the synchronization actions ($a\_wait$ and $a\_not\_avail$) between the audio buffer module and audio manager do not work correctly.

3. Design and Implementation of Media Synchronization Protocols

We assume that each media synchronization protocol is given as concurrent periodic timed automata with temporal properties. We also assume that each protocol is implemented as concurrent programs using multi-threaded mechanisms. In such programs, it takes some variable time for executing the synchronized operations. In the specification, such execution time is described as a parameter. Such parameter values may have to be decided depending on a given execution environment. On the other hand, there exist some parameters whose values can be decided in the design phase. For example, stream decoding time can be decided in advance if we use a hardware decoder.

There exist a lot of combinations of the parameter values which make the given concurrent periodic timed automata behave correctly. In addition, the values of
the parameters strongly depend on each other. Thus the protocol with given parameter values may not behave correctly or may not have enough performance in a certain execution environment. For example, if the time interval assigned for the execution of a synchronized operation is too short, the protocol may not execute the operation. By contrast, if the assigned time for some operations is too large, the succeeding operations may not be able to have finished within the specified time period.

In this paper, we decide the values of parameters using a *parametric model checking* method and *functional testing*. A parametric model checking method derives parameter conditions for a given specification (a concurrent periodic timed automaton) and temporal properties. If the values of parameters satisfy the derived conditions, then the given concurrent periodic timed automaton satisfy the given temporal properties.

Next we select a combination of parameter values according to the derived conditions, and develop an IUT (Implementation Under Test) based on the selected parameter values. We check whether the IUT can behave correctly in a certain environment (*functional testing*). If the IUT does not pass the functional testing, we decide that the IUT does not execute the target function in the execution environment and we adjust the parameter values to make IUT execute the function. Even if the IUT has passed the functional testing, there may exist a case that the protocol does not have enough performance and/or does not have enough time intervals for some actions. In that case, we also find other suitable values for the corresponding parameters to improve on the performance. We repeat such operation for some times. If the IUT passes the functional testing and has enough performance, then we decide to stop the functional testing. Otherwise, i.e. the IUT does not pass the functional testing or does not have enough performance, we select another combination of parameter values based on the derived parameter conditions. Hence, we can find appropriate parameter values for a given execution environment.
Table 1: Syntax of RPCTL

\[ f ::= \text{true} \quad \text{(universally valid)} \]

\[ f \lor f \quad \text{(disjunction)} \]

\[ \neg f \quad \text{(negation)} \]

\[ f \Rightarrow f \quad \text{(implication)} \]

\[ \langle a \rangle \sim_p f \quad \text{(existential ‘next’ operator)} \]

\[ [a] \sim_p f \quad \text{(universal ‘next’ operator)} \]

\[ f \mathbf{EU}_p f \quad \text{(existential ‘until’ operator)} \]

\[ f \mathbf{AU}_p f \quad \text{(universal ‘until’ operator)} \]

\[ EF_p f \quad \text{(existential ‘eventually’ operator)} \]

\[ AF_p f \quad \text{(universal ‘eventually’ operator)} \]

\[ EG_p f \quad \text{(existential ‘always’ operator)} \]

\[ AG_p f \quad \text{(universal ‘always’ operator)} \]

4. Parametric Model Checking

Parametric model checking methods derive parameter conditions for a state transition model to satisfy given temporal properties. If the derived parameter conditions are satisfied by the parameter values, then the temporal properties hold on the state transition model.

In [4], we have proposed a parametric model checking method for periodic timed automata. Since the model is restricted to be periodic, our method can derive parameter conditions efficiently by analyzing at most three periods’ behavior of a given periodic timed automaton (see [4] for details). In our method, temporal properties are written in RPCTL (Real-time and Parametric extension of Computation Tree Logic) [4]. RPCTL is a logic to specify temporal properties at the state of a timed automaton for its succeeding behavior using temporal operators with timing constraints which may contain parameters.

**Definition 1.** The syntax of RPCTL formula is defined by the BNF shown in Table 1, where \( a \in A \) is an action name, \( p \) is a linear expression which may contain...
A Method for Designing Multimedia Protocols using Both Parametric Model Checking and Functional Testing

parametric variables and \( \sim \in \{<, \leq, =, \geq, >\} \) is a comparison operator. We may omit ‘\( \sim p \)’ specifier in that case ‘\( \geq 0 \)’ is assumed.

Intuitive meaning of basic constructs of RPCTL is as follows. ‘true’ holds at any concrete state. ‘false’ never holds at any concrete state, which is equivalent to ‘\( \neg \)true’. ‘\( f_1 \land f_2 \)’ holds if and only if both \( f_1 \) and \( f_2 \) hold. ‘\( f_1 \lor f_2 \)’ and ‘\( f_1 \Rightarrow f_2 \)’ are also defined similarly to propositional logic. ‘\( (a)\leq pf \)’ holds at \((s, \sigma)\) if and only if there exists some transition from \((s, \sigma)\) performing \( a \) within \( p \) time units, such that \( f \) holds at the next state. Since we can define similarly if \( \sim \) is other than \( \leq \)(case of \(<, =, \geq, >\)), we only mention the case of \( \leq \) in the followings. ‘\( [a]\leq pf \)’ holds if and only if for any transition from the state performing \( a \) within \( p \) time units, \( f \) holds at the next state, which is the same as ‘\( \neg (a)\leq pf \)’. ‘\( f_1 E U \leq pf_2 \)’ holds if and only if there exists some transition sequence such that \( f_2 \) eventually holds within \( p \) time units and until then, \( f_1 \) always holds. ‘\( f_1 A U \leq pf_2 \)’ holds if and only if for any transition sequence, \( f_2 \) eventually holds within \( p \) units of time and until then, \( f_1 \) always holds. We also use ‘\( E F \leq pf \)’, ‘\( A G \leq pf \)’, ‘\( A F \leq pf \)’ and ‘\( E G \leq pf \)’ as abbreviations for ‘\( true E U \leq pf \)’, ‘\( \neg EF \leq pf \)’, ‘\( true A U \leq pf \)’ and ‘\( \neg AF \leq pf \)’, respectively.

**Example 4.1.** For the media synchronization protocol, we consider the property: “the protocol must return to the initial state by executing return transition within the given period \( T \).” If the property is satisfied, then the protocol executes a periodic behavior. In RPCTL, the property is described as the following formula.

\[
AG\leq T(\neg EF \leq 0((return) \land true))
\]

In this paper, we describe the specification of a given protocol as concurrent periodic timed automata with temporal properties given as a set of RPCTL formulas. A predicate \( F(p_1, \ldots, p_k) \) can be obtained by executing the parametric model checking [4] for a given specification and RPCTL formulas. Here, \( p_1, \ldots, p_k \) are parameters which are contained in the specification. \( F(v_1, \ldots, v_k) \) is true if and only if the specification with the concrete parameter values \( v_1, \ldots, v_k \) can execute
their transitions satisfying the given RPCTL formulas and return to the initial state periodically.

Since all parameters are treated as real numbers and only linear inequalities can be used for describing the transition conditions, we will construct $F(p_1, \ldots, p_k)$ as a predicate consisting of linear inequalities of parameters $p_1, \ldots, p_k$.

If $F(p_1, \ldots, p_k)$ is unsatisfiable, i.e., a parametric model checker returns $false$ as a model checking result, we conclude that the specification cannot satisfy the temporal properties and the specification is wrong.

**Example 4.2.** The following predicate is obtained for the media synchronization protocol in Fig. 2.2 and RPCTL formula in Example 4.1 by executing parametric model checking. Here, for simplicity of discussion, we consider only the audio buffer module and audio manager.

(1) $pm_2 < pb_1$

(2) $pb_1 + pb_2 < pm_1$

(3) $pb_2 \leq pb_3$

(4) $pb_2 + pb_5 \leq T$

(5) $pb_2 + pb_4 + pb_5 \leq T$

(6) $pm_3 \leq pm_4$

(7) $pm_5 \leq pm_6$

(8) $pm_3 + pm_7 \leq T$

(9) $pm_5 + pm_7 \leq T$

(10) $pm_2 + pm_5 + pm_7 \leq T$

If Condition (1) is satisfied, then action $a_{\text{not avail}}$ become available in the audio manager before action $a_{\text{not avail}}$ become available in the audio buffer module. As a result, $a_{\text{not avail}}$ can be executed synchronously. Condition (2) denotes that action $a_{\text{wait}}$ in audio manager can wait for action $a_{\text{wait}}$ to become executable in audio buffer module. If Condition (2) is satisfied, then $a_{\text{wait}}$ can be executed synchronously.

SIU 2004
Figure 4.3: The composed machine of audio buffer module and audio manager

Condition (3) denotes that there exist an interval for action $a_{\text{wait}}$! in the audio buffer module. In addition, conditions (6) and (7) represent that there exist intervals for actions $\text{play}!$ from state 22 to state 24 and $\text{play}!$ from state 23 to state 24, respectively.

Conditions (4) and (5) denote that the audio buffer module can execute action $\text{return}!$ within $T$ time units. Conditions (8), (9) and (10) represent that the audio manager can execute action $\text{return}!$ within $T$ time units.

If the above conditions hold, then the IUT can return to its initial state by executing action $\text{return}!$ within $T$ time units.

In order to derive parameter conditions, we construct the composed machine for the given concurrent periodic timed automata. Here, for simplicity of discussion, we construct the composed machine of the audio buffer module and audio manager in Fig. 2.2, which is shown in Fig. 4.3. Each state has the reset transitions $\text{reset}$, which are not shown in Fig. 4.3. The audio buffer module has four states and the audio manager has six states. Thus the composed machine has 24 states as their combinations. However, the number of reachable states from the initial state (10, 20) of the composed machine is seven if we consider synchronized transitions. We expect that the composed machine can return to the initial state (10, 20) by executing $\text{return}$ transition at (13, 25), and that reset transition $\text{reset}$ does not be executed in any cases.

Our model checking method [4] constructs the composed machine and derives a parameter conditions by depth-first search using the on-the-fly technique. Although we use the method in [4], other model checkers such as Refs. [5, 6, 7] can be also used. Some class restrictions may be given depending on the model.

5. Functional Testing

5.1. Functional Testing Problem

Functional testing checks whether a given IUT can correctly execute test sequences in a given execution environment or not. Each test sequence corresponds to a function described in the specification.

In this paper, we propose two testing methods Test (i) and Test (ii). Test (i) is used for deciding suitable parameter values for a given execution environment. We decide parameter values based on the parameter conditions obtained by the parametric model checking in Section 4. Test (ii) is used for checking whether the IUT can tolerate for various CPU load conditions of the execution environment. Here, we assume that the IUT has already passed the conformance testing using, for example, [10, 11, 12, 13]. In addition, we also assume that an execution environment does not affect the temporal order of actions but affect the execution timing of actions.

5.1.1. Testing Method for Test (i)

In Test (i), we decide adequate values for parameters which are contained in a given protocol specification. Inputs of Test (i) are a concurrent periodic timed automaton (protocol specification) $M$ which does not have concrete parameter values and a predicate $F(p_1, \ldots, p_k)$ which represents parameter conditions obtained by executing parametric model checking. An output of Test (i) is a set of parameter values, i.e. $(p_1, \ldots, p_k) = (v_1, \ldots, v_k)$ where $v_1, \ldots, v_k$ are concrete values.

First, we decide an initial set of parameter values $(v_1, \ldots, v_k)$ by solving the parameter conditions which are obtained in the model checking phase. Since the
A Method for Designing Multimedia Protocols using Both Parametric Model Checking and Functional Testing

parameter conditions consist of linear inequalities, we can obtain a set of parameter values.

Next, we decide test sequences. We explain the details in Section 5.2. Then, we execute the test sequences. In order to execute each test sequence, we must decide the execution timing of actions in the test sequence. The method to decide concrete execution timing and the way to make the test sequence execute at the specified timing are explained in Sections 5.3 and 5.4, respectively. In real-time environments with variable CPU loads and/or network environments, a given test sequence may be executed in some case. However, it may not be able to be executed in some other case. We execute each test sequence for m times, and count the number where the test sequence has been executed satisfying the given temporal constraints. If an action of the test sequence cannot be executed in the specified timing, then we record its difference between the expected execution timing and actual execution timing.

Finally, we evaluate the results. Suppose that the test sequence can be executed correctly for n times. If n/m ≥ γ, then we conclude that the given IUT can execute the test sequence correctly and has passed the functional testing. The value of γ (0 ≤ γ ≤ 1) depends on the specification. In most cases, the value of γ is about one. If n/m < γ, then we decide that the IUT fails the functional testing. In this case, we modify the parameter values based on the time difference obtained in the execution of the test sequence.

Assume that the transition condition for transition $g$ is $g_{\text{min}} \leq t_g \leq g_{\text{max}}$, where $t_g$ denotes the execution time of transition $g$, and $g_{\text{min}}$ and $g_{\text{max}}$ are the parameter values decided in the above phase, i.e. $g_{\text{min}} = v_i$ and $g_{\text{max}} = v_j$. We execute a given test sequence for m times, and obtain the set of the actual execution times $t_{g1}, \ldots, t_{gm}$ of each transition $g$. Here, suppose that $g_{\text{min}_\text{act}}$ and $g_{\text{max}_\text{act}}$ denote the minimum and maximum values in $\{t_{g1}, \ldots, t_{gm}\}$, respectively. In this case, for example, we modify parameters $g_{\text{min}}$ and $g_{\text{max}}$ to $g_{\text{min}} + \lambda(g_{\text{min}_\text{act}} - g_{\text{min}})$ and $g_{\text{max}} + \lambda(g_{\text{max}_\text{act}} - g_{\text{max}})$, respectively. Here ($\lambda > 0$). Then we check whether the new set of parameter values $(v'_1, \ldots, v'_k)$ can satisfy the predicate $F(p_1, \ldots, p_k)$. If the designers want to make the executable interval for transition $g$ much wider, we may give a larger value for $\lambda$. Then the executable interval
of transition \( g \) becomes wide. However, note that if we do so, the executable intervals for the succeeding transitions in the test sequence may be narrow.

If the new set satisfies the predicate \( F(p_1, \ldots, p_k) \), then we repeat the above functional testing. Otherwise, we modify the parameter values again and check whether the another set of parameter values satisfy the predicate \( F(p_1, \ldots, p_k) \). If we can not find a set of parameter values satisfying \( F(p_1, \ldots, p_k) \), we select one of followings: (1) we return to the first step and decide a initial set of parameter values again, (2) we conclude that the IUT fails the functional testing.

After the IUT has passed the above Test (i), we check whether the IUT can tolerate for various variation of the execution environment in Test (ii).

5.1.2. Testing Method for Test (ii)

We check whether the IUT can tolerate for various CPU load conditions of the execution environment in Test (ii). Inputs of Test (ii) are a concurrent periodic timed automaton \( M \) which has concrete parameter values and load graphs which describe a CPU load at each time. An output of Test (ii) is set of the ratio of the number of success executions and the number of test executions.

First, we decide test sequences for Test (ii). The test sequences may be the same as those of Test (i).

Next, we execute the test sequences under various execution environments. According to the load graphs, we control a CPU load. The method to control execution environments such as CPU loads is described in Section 5.4. We count the number where the test sequence can be executed satisfying the time constraints. If the success rate in various execution environments exceeds a threshold, then we decide that the given parameter values are suitable for the IUT to behave correctly in various load environments.

5.2. Decision of Testing Functions and Their Test Sequences

For functional testing, first, we must decide functions which we intend to test. Then, we generate a set of sequences \( \Psi = \{ \psi_1, \psi_2, \ldots, \psi_k \} \) for testing a function in the given specification \( M = \{ M_1, M_2, \ldots, M_k \} \). Here, each \( \psi_i \) consists of a
sequence of transitions on $M_i$ in $M$ ($1 \leq i \leq k$). Since $M$ consists of $k$ periodic timed automata, $k$ sequences are executed in parallel on $M$ for each functional testing. For functional testing, we may generate several sets of sequence $\Psi$.

**Example 5.1.** For our media synchronization protocol $M = \{\text{audio buffer module, audio manager, video buffer module, video manager, synchronizer}\}$, we consider the following functional tests: (a) test whether the specified executable time intervals are suitable for the execution of the synchronous operations between the audio manager and video manager, (b) test whether the protocol’s behavior is suitable for the case that the expected audio/video frame is not received.

Here, we consider functional test (a). For functional test (a), we generate test sequences which contain action play. In our media synchronization protocol, audio manager, video manager and synchronizer execute action play synchronously. Action play, naturally, denotes the synchronous operation between audio manager and video manager. We generate sequences which contain action play for these three modules. For other modules (audio buffer module and video buffer module), we generate sequences such that the above three sequences can be executed. The following five transition sequences correspond to the test sequences for audio buffer module, audio manager, video buffer module, video manager and synchronizer, respectively.

- **audio buffer module**: $a\_\text{wait}\? \ a\_\text{wait}\! \ a\_\text{ready}\! \ return\!
- **audio manager**: $a\_\text{wait}\? \ a\_\text{ready}\? \ play\! \ a\_\text{play}\_\text{end}\! \ return\!
- **video buffer module**: $v\_\text{wait}\? \ v\_\text{wait}\! \ v\_\text{ready}\! \ return\!
- **video manager**: $v\_\text{wait}\? \ v\_\text{ready}\? \ play\! \ v\_\text{play}\_\text{end}\! \ return\!
- **synchronizer**: $\text{play}\?

In this phase, we do not have to consider whether the set of the generated test sequences are executable in parallel. In the next phase, we derive the executable time interval for every action in each test sequence using the linear programming technique. If we cannot find the executable time interval for some action of a test sequence, then we generate another test sequence for the function.

SIU 2004
5.3. Decision of Execution Timing

The systems considered in this paper are real-time systems. When we execute test sequences for real-time systems, we must consider the execution timing for the transitions in the test sequences. However, since outputs are given from IUTs and uncontrollable, we cannot designate their output timing in advance. Also their output timing affects the executable timing for the succeeding I/O actions in the test sequence. Therefore, in general, the executable timing of each input action in a test sequence can be specified by a function of the execution time of the preceding I/O actions. In order to decide execution timings, we introduce the symbolic trace [14, 16].

For each function which we intend to test, we generate a set of sequences $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$ described in the previous section. First, we translate each transition sequence $\psi_i$ to the symbolic trace. Here, we explain its intuitive meaning. The formal definition of the symbolic trace is given in [14, 16].

5.3.1. Symbolic Trace

A symbolic trace is another representation of a transition sequence. We introduce a global clock $c_g$ and represent transition conditions using only the execution time of actions in the transition sequence. The global clock $c_g$ is reset when the timed automaton starts and returns to the initial state every time period $T$. 

SIU 2004
For example, Fig. 5.4 (i) denotes a transition sequence of audio buffer module. We assume that \(t_g\) denotes the value of global clock \(c_g\). We also suppose that \(e_1, e_2, e_3\) and \(e_4\) denote the global execution time for \(a\_avail\), \(a\_wait\), \(a\_ready\) and \(return\), respectively.

First, the value of \(tab_1\) is equivalent to time \(t_g\). Since clock \(tab_1\) is reset to zero when \(a\_avail\) is executed, the value of \(tab_1\) becomes \(t_g - e_1\) after \(a\_avail\) is executed. The original transition condition for \(a\_wait\) is \(pb_2 < tab_1 < pb_3\). Since \(tab_1 = t_g - e_1\), the condition is translated into \(pb_2 < t_g - e_1 < pb_3\). \(t_g\) represents the value of global clock \(c_g\) and \(e_2\) denotes the global execution time of \(a\_wait\). Hence, \(t_g\) is equivalent to \(e_2\). Therefore, \(pb_2 < e_2 - e_1 < pb_3\) is obtained as shown in Fig. 5.4 (ii). And then, the value of \(tab_1\) becomes \(t_g - e_2\) after \(a\_wait\) is executed. In Fig. 5.4 (ii), the transition conditions are described using \(t_g, e_1, e_2, e_3\) and \(e_4\) where the value of \(tab_1\) is also shown.

In Fig. 5.4 (iii), we represent those transition conditions using only the execution time \(e_1, e_2, e_3\) and \(e_4\) of the four actions. These conditions are obtained by transforming the conditions in Fig. 5.4 (ii). This transition sequence in Fig. 5.4 (iii) represents the symbolic trace for the transition sequence in Fig. 5.4 (i).

5.3.2. Execution Timing Using Linear Programming Techniques

A symbolic trace contains the execution time for each transition in the trace. We derive an execution time interval for each transition by using linear programming techniques. We assume that we derive execution time intervals for all transitions in a set of symbolic traces \(\Xi = \{\xi_1, \xi_2, \ldots, \xi_k\}\), where a symbolic trace \(\xi_i = t_{i,1}t_{i,2}\cdots t_{i,l} (1 \leq i \leq k)\) and \(t_{i,j} = (u_{i,j}, a_{i,j}, e_{i,j}, \phi_{i,j}, v_{i,j}) (1 \leq j \leq l)\). Here, \(u_{i,j}\) and \(v_{i,j}\) denote states, \(a_{i,j}, e_{i,j}\) and \(\phi_{i,j}\) represent action name, execution time and time constraints, respectively.

We introduce two new variables \(e_{i,j,min}\) and \(e_{i,j,max}\) for representing the interval of the execution time \(e_{i,j}\) of each transition \(t_{i,j}\) in the symbolic trace \(\xi_i\). Here, \(e_{i,j,min}\) and \(e_{i,j,max}\) represent the earliest and latest execution time of \(t_{i,j}\) respectively. That is, \(e_{i,j,min}\) and \(e_{i,j,max}\) represents the executable time interval for transition \(t_{i,j}\). We construct the following conditions which contain these variables.

SIU 2004
• Transition order conditions.

To execute each transition in the expected order, we construct the following condition for each symbolic trace $\xi_i$.

$$e_{i,1,min} \leq e_{i,1,max} \leq e_{i,2,min} \leq e_{i,2,max} \leq \cdots \leq e_{i,l,min} \leq e_{i,l,max}$$

• Synchronized transition conditions.

If the $x$th transition in the symbolic trace $\xi_i$ and the $y$th transition in the symbolic trace $\xi_j$ must be executed synchronously, then we construct the following condition.

$$e_{i,x,min} = e_{j,x,min} \land e_{i,x,max} = e_{j,y,max}$$

• Time constraints conditions.

For each time constraint in the symbolic trace $\xi_i$ whose form is $e_{i,j} \sim f$ where $\sim \in \{<, \leq, ==, \geq, >\}$ and $f = a_1 e_{i,1} + \cdots + a_n e_{i,n} + a_{n+1} (n < j)$, we construct the following conditions.

$$e_{i,j,max} \sim a_1 e_{i,1,min} + \cdots + a_l e_{i,l,min} + a_{l+1} \quad (\sim \in \{<, \leq, ==\})$$

$$e_{i,j,min} \sim a_1 e_{i,1,max} + \cdots + a_l e_{i,l,max} + a_{l+1} \quad (\sim \in \{\geq, >\})$$

• An objective function.

To make the time interval of each transition as wide as possible, the following objective function is introduced.

$$\sum w_{i,j}(e_{i,j,max} - e_{i,j,min})$$

Each weight $w_{i,j}$ is decided by the testers. Since the testers cannot control the output timing from the IUT, the testers should generally assign larger weights for output transitions than input transitions.

We construct a linear programming problem representing the above conditions and solve it. Note that some linear programming solvers may return a solution such that an interval for one transition is very large and intervals for other transitions are very small. Thus we may give additional constraints for intervals such that $e_{i,j,max} - e_{i,j,min} \geq const_{i,j}$ where $const_{i,j}$ denotes the minimum time interval required for $t_{i,j}$. 

SIU 2004
We derive various execution timings for test sequences and execute functional testing. If we cannot obtain a solution, then we change the value of $\text{const}_{i,j}$ or return to the previous phase and construct another set of test sequences $\Psi$.

**Example 5.2.** We derive time intervals for the sequence in Example 5.1. We assign the following values for the parameters. Here, we describe parameters which are contained the sequence. Since the size of the video stream is generally larger than that of an audio stream, we assign larger values to the parameters corresponding to video streams.

\[
\begin{align*}
p_{b1} &= 5, \quad p_{b2} = 3, \quad p_{b3} = 15, \quad p_{b4} = 20, \quad p_{b5} = 5, \\
p_{m1} &= 15, \quad p_{m3} = 2, \quad p_{m4} = 20, \quad p_{m7} = 3, \\
q_{b1} &= 10, \quad q_{b2} = 10, \quad q_{b3} = 20, \quad q_{b4} = 20, \quad q_{b5} = 10, \\
q_{m1} &= 20, \quad q_{m3} = 4, \quad q_{m4} = 20, \quad q_{m7} = 3, \\
T &= 50
\end{align*}
\]

The above parameter values satisfy the parameter condition obtained in model checking phase. We add constraints such that each transition has at least five time units interval, i.e. $e_{i,j,\text{max}} - e_{i,j,\text{min}} \geq 5$.

According to the weight for each transition interval, we can obtain various test sequences. First, we assign the same weight for all transition intervals and obtain the following result. The result represents action names and its time intervals. Time intervals are described below action names.

\[
\begin{align*}
\text{audio buffer module} & : a_{\text{avail}}? \quad a_{\text{wait}}! \quad a_{\text{ready}}! \quad \text{return!} \\
& \quad 0 - 5 \quad 8 - 14 \quad 14 - 27 \quad 42 - 50 \\
\text{audio manager} & : a_{\text{wait}}? \quad a_{\text{ready}}? \quad \text{play!} \quad a_{\text{play\_end}}! \quad \text{return!} \\
& \quad 8 - 14 \quad 14 - 27 \quad 29 - 34 \quad 37 - 42 \quad 42 - 50 \\
\text{video buffer module} & : v_{\text{avail}}? \quad v_{\text{wait}}! \quad v_{\text{ready}}! \quad \text{return!} \\
& \quad 0 - 5 \quad 15 - 20 \quad 20 - 25 \quad 42 - 50 \\
\text{video manager} & : v_{\text{wait}}? \quad v_{\text{ready}}? \quad \text{play!} \quad v_{\text{play\_end}}! \quad \text{return!} \\
& \quad 15 - 20 \quad 20 - 25 \quad 29 - 34 \quad 37 - 42 \quad 42 - 50 \\
\text{synchronizer} & : \text{play?} \\
& \quad 29 - 34
\end{align*}
\]
If we assign larger weights to some actions, then we can obtain another time intervals. Here, we consider actions $a_{\text{play}}\cdot \text{end!}$ and $v_{\text{play}}\cdot \text{end!}$, which are output actions and they are not synchronized actions. We assign larger weights to these actions. The result is given as follows.

**audio buffer module** :

$$a_{\text{avail}}? \rightarrow a_{\text{wait}}! \rightarrow a_{\text{ready}}! \rightarrow \text{return!}$$

$$0 \rightarrow 5 \quad 8 \rightarrow 14 \quad 14 \rightarrow 27 \quad 50 \rightarrow 50$$

**audio manager** :

$$a_{\text{wait}}? \rightarrow a_{\text{ready}}? \rightarrow \text{play!} \rightarrow a_{\text{play}}\cdot \text{end!} \rightarrow \text{return!}$$

$$8 \rightarrow 14 \quad 14 \rightarrow 27 \quad 29 \rightarrow 34 \quad 37 \rightarrow 50 \quad 50 \rightarrow 50$$

**video buffer module** :

$$v_{\text{avail}}? \rightarrow v_{\text{wait}}! \rightarrow v_{\text{ready}}! \rightarrow \text{return!}$$

$$0 \rightarrow 5 \quad 15 \rightarrow 20 \quad 20 \rightarrow 25 \quad 50 \rightarrow 50$$

**video manager** :

$$v_{\text{wait}}? \rightarrow v_{\text{ready}}? \rightarrow \text{play!} \rightarrow v_{\text{play}}\cdot \text{end!} \rightarrow \text{return!}$$

$$15 \rightarrow 20 \quad 20 \rightarrow 25 \quad 29 \rightarrow 34 \quad 37 \rightarrow 50 \quad 50 \rightarrow 50$$

**synchronizer** :

$$\text{play?}$$

$$29 \rightarrow 34$$

We can obtain wider intervals for actions $a_{\text{play}}\cdot \text{end!}$ and $v_{\text{play}}\cdot \text{end!}$. Instead of that, the intervals for action $\text{return!}$ for all modules become narrow. However, this is a more reasonable solution.

On the other hand, we can modify parameter values as the initial values, since there exist some parameter values which satisfy the parameter conditions obtained in the model checking phase. Here, we assign one and eight for $pb_2$ and $qb_2$, respectively. The new parameter values still satisfy the parameter conditions. We obtain the following result.

**audio buffer module** :

$$a_{\text{avail}}? \rightarrow a_{\text{wait}}! \rightarrow a_{\text{ready}}! \rightarrow \text{return!}$$

$$0 \rightarrow 5 \quad 6 \rightarrow 12 \quad 12 \rightarrow 25 \quad 50 \rightarrow 50$$

**audio manager** :

$$a_{\text{wait}}? \rightarrow a_{\text{ready}}? \rightarrow \text{play!} \rightarrow a_{\text{play}}\cdot \text{end!} \rightarrow \text{return!}$$

$$6 \rightarrow 12 \quad 12 \rightarrow 25 \quad 27 \rightarrow 32 \quad 35 \rightarrow 50 \quad 50 \rightarrow 50$$

**video buffer module** :

$$v_{\text{avail}}? \rightarrow v_{\text{wait}}! \rightarrow v_{\text{ready}}! \rightarrow \text{return!}$$

$$0 \rightarrow 5 \quad 13 \rightarrow 18 \quad 18 \rightarrow 23 \quad 50 \rightarrow 50$$

**video manager** :

$$v_{\text{wait}}? \rightarrow v_{\text{ready}}? \rightarrow \text{play!} \rightarrow v_{\text{play}}\cdot \text{end!} \rightarrow \text{return!}$$

$$13 \rightarrow 18 \quad 18 \rightarrow 23 \quad 27 \rightarrow 32 \quad 35 \rightarrow 50 \quad 50 \rightarrow 50$$

**synchronizer** :

$$\text{play?}$$

$$27 \rightarrow 32$$

SIU 2004
In this case, the time intervals between actions $a_{\text{avail}}?.$ and $a_{\text{wait}}!.$ in the audio buffer module and actions $v_{\text{avail}}?.$ and $v_{\text{wait}}!.$ in the video buffer module are shorter than the previous case. In addition, we can obtain wider intervals for actions $a_{\text{play.end}}!.$ and $v_{\text{play.end}}!.$ Instead, the executable intervals for their preceding actions become earlier.

5.4. Test Execution and Testing Environment

We consider a testing environment [16] which contains a load controller (Fig. 5.5). Our model, concurrent periodic timed automata, contains both input transitions and output transitions. The tester can control the timing to give an input to the IUT. On the other hand, the tester cannot control the timing to receive an output from the IUT. In Test (i) and Test (ii), the tester control the output timing using load controller.

The load controller gives additional CPU loads according to the requests from the tester. The tester uses a load graph to control the CPU loads. A load graph describe a CPU load at each time. If the IUT does not return the outputs at the expected time, then the tester modifies the load graph and executes the test sequence again. If the given test sequence cannot be executed correctly for several CPU loads, then we conclude that the IUT cannot adequately execute the test sequence.

SIU 2004
6. Conclusions

In this paper, we propose a method to design media synchronization protocols using parametric model checking and functional testing. The proposed method generally can be applied to real-time protocols with time constraints. In our method, for a given specification with temporal properties, we derive the parameter condition by applying parametric model checking method. Based on the derived condition, we obtain adequate parameter values by executing functional testing for several times.

One of our future work is to introduce a statistical approach for evaluating result of functional testing. And we plan to apply the proposed method to several practical protocols.

References


SIU 2004


Authors addresses:
Takanori Mori.
Communications Research Laboratory,
Koganei, Tokyo 184-8795, Japan.
t-mori@crl.go.jp

Akio Nakata.

SIU 2004
Graduate School of Information and Science Technology, Osaka University,
Toyonaka, Osaka 560-8531, Japan.
nakata@ist.osaka-u.ac.jp

Teruo Higashino.
Graduate School of Information and Science Technology, Osaka University,
Toyonaka, Osaka 560-8531, Japan.
higashino@ist.osaka-u.ac.jp

SIU 2004