PAPER *Special Section on Mathematical Systems Science and its Applications*

# Refinement and Verification of Sequence Diagrams Using the Process Algebra CSP

**Tomohiro KAIZU**[†a], ***Nonmember***, **Yoshinao ISOBE**[††b], ***and*** **Masato SUZUKI**[†c], ***Members***

**SUMMARY** Sequence diagrams are often used in the modular design of softwares. In this paper, we propose a method to verify correctness of sequence diagrams. With this method, using the process algebra CSP, concurrent systems can be synthesized from a number of sequence diagrams. We define new CSP operators for the synthesis of sequence diagrams. We also report on a tool implementing our synthesis method and demonstrate how the tool analyzes sequence diagrams.

***key words:*** *sequence diagram, process algebra, CSP, process synthesis*

## 1. Introduction

The requirements for the software systems become more complicated and diversified in recent years. To implement such complex systems, component-based programming has spread.

UML diagrams are often used for designing software components. UML is a standardized modeling language developed by OMG. Especially in upstream development, UML sequence diagrams are frequently used to understand and verify the behavior of components.

However, the UML specification is complicated and flexible. So it is difficult to verify UML diagrams automatically. It has relied on manual review to find mistakes such as inconsistencies and insufficient refinements between sequence diagrams. If such mistakes are found in a late development stage, it may take a lot of time and cost to correct them.

In this paper, we define a subset of sequence diagrams with formal semantics and propose a method to verify correctness of the sequence diagrams. With this method, developers can clarify the specifications by using formal description and find bugs by using automatic verification.

Compared with the related works described in Sect. 7, the main advantage of this work is nondeterminism can be considered. It means that our approach can handle abstract sequence diagrams. Sequence diagrams are often abstract in early development stage. Our approach can be applied to such diagrams.

To verify sequence diagrams, we propose a synthesis method of a formal expression called CSP (Communicating Sequential Processes) [1], [2] from sequence diagrams in order to find mistakes in the early design stage based on the process algebra CSP. This synthesis method consists of two steps: At first, an order of sending and receiving is extracted from a sequence diagram for each component and it is formally expressed as a CSP process. Next, two or more CSP processes extracted from a number of sequence diagrams for the component is combined to a CSP process which represents the whole behavior of the component. This synthesis method allows us to verify properties of the concurrent system consisting of the components by using CSP-tools, for example, the model checker FDR [3].

The paper is organized as follows: First, we briefly explain sequence diagram. In Sect. 3, we introduce CSP and give new operators $\circ$ and \$ for combining two or more sequence diagrams. Then, the synthesis method is presented. In Sects. 4 and 5, we report on a sequence diagram synthesizer which is an implementation of our synthesis method and demonstrates the tool by a shopping site example. Finally, in Sect. 6, we discuss related works.

## 2. Sequence Diagrams

Sequence diagram is one of the diagrams defined in UML, which represents the flow of messages between objects chronologically. UML is a specification language for object modeling developed and standardized by OMG.

A sequence diagram consists of lifelines, messages, activations, and state invariants. Figure 1 is an example of a sequence diagram.

Each element is explained as follows.

- Lifeline: A lifeline is described in a dotted line, and

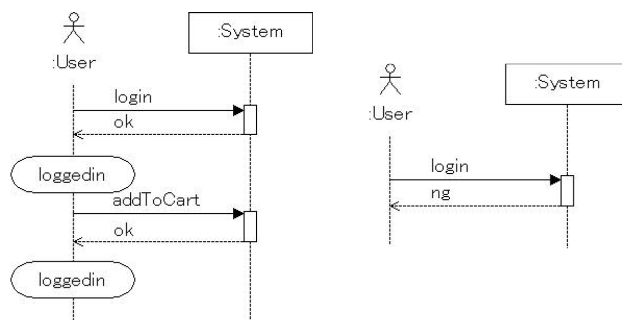


**Fig. 1** An example of sequence diagrams.

Manuscript received March 27, 2012.
Manuscript revised August 21, 2012.
[†]The authors are with Japan Advanced Institute of Science and Technology, Nomi-shi, 923-1211 Japan.
[††]The author is with National Institute of Advanced Industrial Science and Technology, Tsukuba-shi, 305-8568 Japan.
a) E-mail: tkaizu@jaist.ac.jp
b) E-mail: y-isobe@aist.go.jp
c) E-mail: suzuki@jaist.ac.jp
DOI: 10.1587/transfun.E96.A.495

Copyright © 2013 The Institute of Electronics, Information and Communication Engineers

shows the existence of the object. At the head of the lifeline, the object which the lifeline represents is shown by a rectangle or a stickman shape. The rectangular object shows a component in the system, and the object of the stickman shape shows an actor outside of the system.

- Message: Sent and received messages are described by arrows between lifelines. A return message is described in a dotted arrow. In this paper, we assume that every message is synchronous and it must have a return message.
- Activation: An activation shows the period during which the object is performing a procedure. An activation is described in a thin rectangle on the lifeline.
- State invariant: A state invariant is a runtime constraint on the participants of the interaction. An state invariant is described in a text in curly bracket on the lifeline.

In this paper, the set *Designs* of sequence diagrams is defined as follows.

$$
\begin{aligned}
&Designs = 2^{SDs}\\
&SDs = 2^{Transitions}\\
&Transitions =\\
&\quad Components \times States \times Messages \times States\\
&Messages =\\
&\quad MessageNames \times Components \times Components
\end{aligned}
$$

where *Components* is the set of component names, *States* is the set of state names, and *MessageNames* is the set of message names. A *transition* $(C, S1, M, S2) \in Transitions$ means the component $C$ sends or receives a message described as $M$, and transfers from specific state $S1$ to the next state $S2$. A *message* $(MN, CS, CR) \in Messages$ means the component $CS$ sends a message whose name is $MN$, and the component $CR$ receives it.

States are described as state invariants in sequence diagrams. If there is no state invariant between messages, we define the state as follows:

- The state is an intermediate state if it is on activations.
- Otherwise, the state is the *default* state.

The order of transitions can be decided by the states. Each intermediate states has an unique name, so we can connect an activation from transitions.

For example, Fig. 1 can be formalized as follows.

$$D_1 = \{SD_1, SD_2\} \in Designs$$

$$
\begin{aligned}
&\{SD_1, SD_2\} \subseteq SDs\\
&\qquad SD_1 = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}\\
&\qquad SD_2 = \{T_9, T_{10}, T_{11}, T_{12}\}
\end{aligned}
$$

$$
\begin{aligned}
&\{T_1, T_2, \ldots, T_{12}\} \subseteq Transitions\\
&\qquad T_1 = (User, d, M_1, i_1)\\
&\qquad T_2 = (User, i_1, M_3, loggedin)\\
&\qquad T_3 = (User, loggedin, M_2, i_2)\\
&\qquad T_4 = (User, i_2, M_3, loggedin)\\
&\qquad T_5 = (System, d, M_1, i_3)\\
&\qquad T_6 = (System, i_3, M_3, d)\\
&\qquad T_7 = (System, d, M_2, i_4)\\
&\qquad T_8 = (System, i_4, M_3, d)\\
&\qquad T_9 = (User, d, M_1, i_5)\\
&\qquad T_{10} = (User, i_5, M_4, d)\\
&\qquad T_{11} = (System, d, M_1, i_6)\\
&\qquad T_{12} = (System, i_6, M_4, d)
\end{aligned}
$$

$$
\begin{aligned}
&\{M_1, M_2, M_3, M_4\} \subseteq Messages\\
&\qquad M_1 = Login = (login, User, System)\\
&\qquad M_2 = Add = (addToCart, User, System)\\
&\qquad M_3 = OK = (ok, System, User)\\
&\qquad M_4 = NG = (ng, System, User)
\end{aligned}
$$

where $d$ is the *default* state for each component and $i_1$, $i_2$, $i_3$, $i_4$, $i_5$, and $i_6$ are the intermediate states. $SD_1$ and $SD_2$ are the sets of transitions in the left and right sequence diagrams in Fig. 1, respectively.

With this definition, developers can define mutual recursions using the same state names. For example, the *default* state appears many times in Fig. 1. Connecting all occurrences of the *default* states for each component, we can synthesize a recursive behavior. It helps developers writing formal behavior using sequence diagrams. Note the scope of each state name is restricted in its component. For example, *User*'s *default* state and *System*'s *default* state are treated as different states. We use pairs of component name and state name to define global state names in the sequence diagram.

## 3. Process Algebra CSP

### 3.1 Introduction to CSP

Process algebra is a theory to describe and to analyze concurrent processes. CSP is a fundamental process algebra that has been successfully applied in various areas, for example, train control system and security protocol [1], [2], [4], [5].

Behaviors and structures of concurrent processes can be formally described in CSP, and then the properties, e.g. deadlock-freeness, livelock-freeness, and refinement relations, can be verified by CSP-tools such as the model checker FDR [3].

CSP processes can be expressed with more than 10 operators, but we introduce the sub-calculus of CSP, which is essential for describing sequence diagrams, defined by the following grammar.

$$
\begin{aligned}
P \ ::= \ & a \rightarrow P \mid P \,\Box\, P \mid P \sqcap P \mid P_X\|_Y P \mid P \setminus X\\
& \mid PN
\end{aligned}
$$

where $a$ is an event name, $X$ and $Y$ are sets of events, and $PN$ is a process name defined by the form of $PN = P$.

$$P1 = in \to sync \to sync \to P1$$
$$P2 = sync \to out \to sync \to P2$$
$$SYS = P1_{\{in,sync\}}\|_{\{sync,out\}}P2$$

**Fig. 2** An example of CSP parallel process.

Each operator is explained as follows.

- Prefix: $a \to P$ (`a -> P` in FDR) can perform the event $a$, and thereafter behaves like the process $P$.
- External choice: $P \Box Q$ (`P [] Q` in FDR) is a process that behaviors like $P$ or $Q$. The choice of $P$ or $Q$ depends on the next event. This choice can be controlled from the outside, i.e. the other processes or the environments.
- Internal choice: $P \sqcap Q$ (`P |~| Q` in FDR) is a process that behaviors like $P$ or $Q$. This choice is internally (nondeterministically) decided, and cannot be controlled from the outside.
- Parallel composition: $P_X\|_Y Q$ (`P [X || Y] Q` in FDR) means $P$ (resp., $Q$) can independently perform events in $(X - Y)$ (resp., $(Y - X)$), and $P$ and $Q$ have to synchronize through events in $(X \cap Y)$.
- Hiding: $P \setminus X$ (`P \ X` in FDR) behaves like $P$ except that events in $X$ are hidden.

Figure 2 is an example of CSP parallel process. $P1$ is an process which repeats the sequential execution of *in*, *sync*, and *sync*. $P2$ is an process which repeats the sequential execution of *sync*, *out*, and *sync*. In the parallel process $SYS$, $P1$ and $P2$ synchronize with *sync* only. Therefore, at first $P1$ performs *in* independently, then $P1$ and $P2$ synchronize with *sync*, then $P2$ performs *out* independently, then $P1$ and $P2$ synchronize with *sync* again, and then this behavior is repeated. Hence, $SYS$ behaves like the following sequential process $SYS'$.

$$SYS' = in \to sync \to out \to sync \to SYS'$$

### 3.2 CSP Equivalence and Refinements

There are some well-known models to define equivalence and refinement relations of CSP processes. In this section, we briefly explain the traces model and the failures model.

In the traces model, the equivalence and the refinement relations are defined with the set $traces(P)$ which is the set of traces, i.e. event sequences, that the process $P$ can execute. For example, the set of traces for the operators $\to$, $\sqcap$, and $\Box$ are defined as follows:

$$traces(a \to P) = \{\langle\rangle\} \cup \{\langle a\rangle^\frown s | s \in traces(P)\}$$
$$traces(P \sqcap Q) = traces(P) \cup traces(Q)$$
$$traces(P \Box Q) = traces(P) \cup traces(Q)$$

The trace-equivalence and the trace-refinement are defined as follows:

$$P =_T Q \Leftrightarrow traces(P) = traces(Q)$$
$$P \sqsubseteq_T Q \Leftrightarrow traces(P) \supseteq traces(Q)$$

where $P \sqsubseteq_T Q$ means $Q$ refines $P$.

In the failures model, equivalence and refinement relations are defined with $traces(P)$ and $failures(P)$ which is the set of pairs $(s, X)$, where $s$ is a trace of $P$ and $X$ is the set of events $P$ refuses after the execution of $s$. For example, the set of failures for the operators $\sqcap$ and $\Box$ are defined as follows:

$$failures(P \sqcap Q) = failures(P) \cup failures(Q)$$
$$\begin{aligned} failures(P \Box Q) = \\ &\{(\langle\rangle, X) | (\langle\rangle, X) \in failures(P) \cap failures(Q)\} \\ &\cup\{(s, X) | s \neq \langle\rangle, \\ &(s, X) \in failures(P) \cup failures(Q)\} \end{aligned}$$

where it is important to note the difference of the two failures. This means that the failures model can distinguish determinism and nondeterminism.

The failure-equivalence and the failure-refinement are defined as follows:

$$\begin{aligned} P =_F Q \Leftrightarrow\ & traces(P) = traces(Q) \\ \wedge\ & failures(P) = failures(Q) \\ P \sqsubseteq_F Q \Leftrightarrow\ & traces(P) \supseteq traces(Q) \\ \wedge\ & failures(P) \supseteq failures(Q) \end{aligned}$$

where $P \sqsubseteq_F Q$ means $Q$ refines $P$.

The failures model cannot treat livelock correctly. However, according to our definition of sequence diagram, each transition must have at least one message. States are changed only after sending or receiving a message, so livelock is not generated in sequence diagrams.

## 4. Extended CSP for Sequence Diagrams

### 4.1 Semantics of Sequence Diagrams

The semantics of sequence diagrams can be given with CSP, where each message in sequence diagrams is translated to a CSP event and a sequence is translated to a CSP process. For example, the System component in sequence diagrams given in Fig. 1 can be translated to CSP as follows:

$$\begin{aligned} \mathcal{P}^{seq}_{D_1}(T_5) &= \mathcal{M}(Login) \to \mathcal{P}^{st}_{D_1}(System, i_3) \\ \mathcal{P}^{seq}_{D_1}(T_6) &= \mathcal{M}(OK) \to \mathcal{P}^{st}_{D_1}(System, d) \\ \mathcal{P}^{seq}_{D_1}(T_7) &= \mathcal{M}(Add) \to \mathcal{P}^{st}_{D_1}(System, i_4) \\ \mathcal{P}^{seq}_{D_1}(T_8) &= \mathcal{M}(OK) \to \mathcal{P}^{st}_{D_1}(System, d) \\ \mathcal{P}^{seq}_{D_1}(T_{11}) &= \mathcal{M}(Login) \to \mathcal{P}^{st}_{D_1}(System, i_6) \\ \mathcal{P}^{seq}_{D_1}(T_{12}) &= \mathcal{M}(NG) \to \mathcal{P}^{st}_{D_1}(System, d) \\ \mathcal{M}(Login) &= call_login.User.System \\ \mathcal{M}(Add) &= call_addToCart.User.System \\ \mathcal{M}(OK) &= call_ok.System.User \\ \mathcal{M}(NG) &= call_ng.System.User \end{aligned}$$

where $\mathcal{P}^{st}_D(C, S)$ is a CSP process corresponding to component $C$ in state $S$. $\mathcal{P}^{seq}_D(T)$ is a CSP process corresponding to a sequence $T$, and $\mathcal{M}$ is a mapping from message structures to CSP events.

The process $\mathcal{P}^{seq}_D$ is formally defined as follows.

$$\mathcal{P}^{seq}_D(T) = \mathcal{M}(M) \rightarrow \mathcal{P}^{st}_D(C, S2)$$

where

$$\mathcal{M}(M) = call_MN.CS.CR$$
$$T = (C, S1, M, S2)$$
$$M = (MN, CS, CR)$$

Note that there are several sequences starting from $S2$. $\mathcal{P}^{st}_D(C, S2)$ is a result of merging every sequence starting from $S2$. The detailed definition will be presented in Sect. 4.2. We will define it to realize our intuition.

There are three major expectations for merging sequences. First, the component can perform all original sequences. One of the sequences starting from $S2$ is selected and processed. Second, same transition can appear in different sequence diagrams. If several sequences start from the same state and have the same message, these are representing exactly same behavior. We do not distinguish which sequence is selected. Finally, in an abstract level, the sender should nondeterministically select one message just before it sends the message. For example, in Fig. 1, there are 2 *login* messages. These 2 messages are representing exactly the same message. The diagram represents that sometimes *System* replies *ok* and sometimes replies *ng*. The decision is nondeterministically made by *System* after *login* message is received.

### 4.2 Sequence Diagram Synthesis Operators

According to the expectation given in Sect. 4.1, we define two new sequence diagram synthesis operators: *Sequence Diagram Merging Operator* $\circ$ and *Sending Event Internalization Operator* \$. With these operators, the synthesized process from $P$, $Q$, $R$... is described as $(P \circ Q \circ R...)\$\Sigma!$. Here $\Sigma!$ is a set of events which these components $P$, $Q$, $R$.. send. The operator $\circ$ is used for combining the same events in different sequence diagrams, and the operator \$ is used for internalizing choices of sending events because they should be decided in sender processes without depending on environments.

The grammar of our CSP including the sequence diagram synthesis operators $\circ$ and \$ is defined.

**Definition 1:** Our CSP syntax is defined by

$$C ::= C_X \|_Y C \mid C \setminus X \mid P$$
$$P ::= a \rightarrow P \mid P \,\square\, P \mid P \sqcap P \mid P \circ P \mid P\$X \mid PN$$

where $a$ is an event name, $X$ and $Y$ are sets of events, and $PN$ is a process name defined by $PN = P$.

We separate the parallel and hiding operators from the other operators. It means connections between components are not dynamically changed. It is a future work to extend our method to dynamically changing system structures.

Before giving the formal semantics of the new operators, we briefly explain the expected properties for the synthesis operators.

(1) The synthesized process can perform all original sequences. In other words, following processes are trace equivalent. Note they are not necessarily failure equivalent.

$$(P \,\square\, Q \,\square\, R \ldots) =_T (P \circ Q \circ R \ldots)\$\Sigma!$$

(2) The same event often appears in different sequence diagrams. In this case, the combined process should have the same next state after the event occurs. For example, if $a \rightarrow b \rightarrow P$ and $a \rightarrow c \rightarrow Q$ are the sequences of a component, the component should be able to handle both $b$ and $c$ after $a$.

$$((a \rightarrow b \rightarrow P) \circ (a \rightarrow c \rightarrow Q))\$\Sigma!$$
$$=_F (a \rightarrow (b \rightarrow P) \circ (c \rightarrow Q))\$\Sigma!$$

Two choice operators $\square$ and $\sqcap$ have been given in CSP, but these operators cannot represent such type of synthesis. For general, the following equation is expected to hold.

$$((a \rightarrow P) \circ (a \rightarrow Q))\$\Sigma! =_F a \rightarrow (P \circ Q)\$\Sigma!$$

(3) Different events make choices of behaviors. Here it is important to note the difference between sending and receiving. One of the sending events should be selected by the process sending it, while one of the receiving events should be selected by the other process, thus senders or environments. Therefore, the following equations are expected to hold.

$$((a \rightarrow P) \circ (b \rightarrow Q))\$\Sigma!$$
$$=_F (a \rightarrow P\$\Sigma!) \sqcap (b \rightarrow Q\$\Sigma!)$$

where $a \in \Sigma!, b \in \Sigma!$

$$((a \rightarrow P) \circ (b \rightarrow Q))\$\Sigma!$$
$$=_F (a \rightarrow P\$\Sigma!) \,\square\, (b \rightarrow Q\$\Sigma!)$$

where $a \notin \Sigma!, b \notin \Sigma!$

We define the semantics of $P \circ Q$ and $P\$\Sigma!$ with traces and failures to satisfy these expected properties. To satisfy the expectation (2), operator $\circ$ is defined as combining the same events to one event. To satisfy the expectation (3), operator \$ is defined as internalizing sending events.

The synthesis process $P \circ Q$ is similar to the external choice $P \,\square\, Q$ except that the same events are combined to an event. To do that, if $P$ and $Q$ can execute the same trace $s$ and they do not refuse the same event $a$ after $s$, then $P \circ Q$ does not refuse it either. This meaning of $P \circ Q$ is defined as follows.

**Definition 2:**

$$traces(P \circ Q) = traces(P) \cup traces(Q)$$
$$failures(P \circ Q) = \{(s, X) | \ (s, X) \in failures(P) \cup failures(Q), \ g(s, P) \Rightarrow (s, X) \in failures(P), \ g(s, Q) \Rightarrow (s, X) \in failures(Q)\}$$

where $g(s, P)$ requires that the trace $s$ is not refused by the process $P$, and it is defined as follows.

$$g(\langle\rangle, P) = true$$
$$g(s\hat{}\langle a\rangle, P) = g(s, P) \wedge ((s, \{a\}) \notin failures(P))$$

The associative law and the commutative law for $\circ$ hold.

$$(P1 \circ P2) \circ P3 =_F P1 \circ (P2 \circ P3),$$
$$P1 \circ P2 =_F P2 \circ P1$$

By these laws, we can use the replicated form without respect to the order of synthesis of $n$ processes. We use the following syntax sugars to describe composition of $n$ processes by $\circ$, $\Box$, and $\sqcap$.

$$\bigcirc_{i\in\{0...n\}} @P_i = P_0 \circ P_1 \circ \cdots \circ P_n$$
$$\Box_{i\in\{0...n\}} @P_i = P_0 \Box P_1 \Box \ldots \Box P_n$$
$$\sqcap_{i\in\{0...n\}} @P_i = P_0 \sqcap P_1 \sqcap \ldots \sqcap P_n$$

Then, the following Theorem 1 holds.

**Theorem 1:**

(1) $\bigcirc_{i\in I} @(a \rightarrow P_i) =_F a \rightarrow (\bigcirc_{i\in I} @P_i)$

(2) $(i \neq j \Rightarrow a_i \neq a_j) \Rightarrow$
$\bigcirc_{i\in I} @(a_i \rightarrow P_i) =_F \Box_{i\in I} @(a_i \rightarrow R_i)$

where $I$ is a non-empty finite index set. These equations can be proved by the following lemmas:

(1) $g(s, P) \Leftrightarrow g(\langle a\rangle\hat{}s, a \rightarrow P)$

(2) $a \neq b \Rightarrow g(\langle a\rangle\hat{}s, b \rightarrow P) = false$

Theorem 1(1) can be applied if all events are the same. Theorem 1(2) can be applied if all events are different. In other cases, we can remove $\circ$ from CSP formula using following corollary.

**Corollary 1:**

$$\bigcirc_{i\in I} @(a_i \rightarrow P_i) =_F \Box_{a\in A} @(a \rightarrow P'_a)$$

where

$$A = \{a \mid \exists i.\ i \in I, a_i = a\}$$
$$P'_a = \bigcirc_{\{i\in I \mid a_i = a\}} @P_i$$

Note $P'_a$ contains $\circ$ in its definition. Therefore, this corollary is not enough to define $\circ$. It can be defined by the fixed point of the equation in the corollary, but it is more tractable to use traces and failures like in Definition 2.

Selections of sending events in the synthesized process have to be internalized. It can be realized by the operator $ defined as follows.

**Definition 3:**

$$traces(P\$Z) = traces(P)$$
$$failures(P\$Z) = failures(P) \cup \{(s, X) | \ \exists Y.((s, Y) \in failures(P) \wedge X \subseteq Y \cup Z), \ \exists a.(s\hat{}\langle a\rangle \in traces(P) \wedge a \notin X))\}$$

The condition $(X \subseteq Y \cup Z)$ means events in Z can be refused in $P\$Z$, and the last condition $(a \notin X)$ requires that only one event is not refused at least. Then, the following Theorem 2 holds[†].

**Theorem 2:**

(1) $(A_! = \phi) \Rightarrow$
$(\Box_{a\in A} @(a \rightarrow P_a))\$\Sigma! =_F \Box_{a\in A_?} @(a \rightarrow P_a\$\Sigma!)$

(2) $(A_! \neq \phi \wedge A_? = \phi) \Rightarrow$
$(\Box_{a\in A} @(a \rightarrow P_a))\$\Sigma! =_F \sqcap_{a\in A_!} @(a \rightarrow P_a\$\Sigma!)$

(3) $(A_! \neq \phi \wedge A_? \neq \phi) \Rightarrow (\Box_{a\in A} @(a \rightarrow P_a))\$\Sigma!$
$=_F \sqcap_{a\in A_!} @(a \rightarrow P_a\$\Sigma!) \triangleright \Box_{b\in A_?} @(b \rightarrow P_b\$\Sigma!)$

where $A$ is a set of events, and $A_! = A \cap \Sigma!$, $A_? = A - \Sigma!$, where "$-$" means the difference of sets[††].

In Theorem 2(3), the operator $\triangleright$ represents timeout and it is a syntax sugar defined as follows.

$$P \triangleright Q = (P \Box Q) \sqcap Q$$

It intuitively means that $P \triangleright Q$ now behaves like P and it behaves like Q after a while.

Before defining the operator $ and proving Theorem 2(3), we did not have a clear prediction about the behavior of the mix case of sending and receiving. Theorem 2(3) gives a clear solution to the case.

In addition, some properties of the operators $\circ$ and $ have been proved. For example, the two operators preserve the failures-refinement as follows.

$$P1 \sqsubseteq_F Q1, P2 \sqsubseteq_F Q2 \Rightarrow P1 \circ P2 \sqsubseteq_F Q1 \circ Q2,$$
$$P \sqsubseteq_F Q \Rightarrow P\$Z \sqsubseteq_F Q\$Z$$

Especially, the operator $\circ$ is very carefully defined for preserving the refinement. For example, the condition $s \in traces(P)$ is similar to and easier than $g(s, P)$ used in the definition of $\circ$, and can be used instead for defining a similar operator to $\circ$. However, the similar operator does not preserve the refinement. Using $g(s, P)$ is an important idea in this work.

Using these definitions, corollary and preservation of

[†]Theorem 2 also hold for any set $Z$ instead of $\Sigma!$, but here we give an instance for clarification.

[††]See http://dr.asukaze.net/sd2csp/ for the proof.

the failures-refinement, the original expectations hold as follows.

$$(1)\quad (P \Box Q \Box R \ldots) =_T (P \circ Q \circ R \ldots)\$\Sigma!$$

Proof: $traces(P \Box Q) = traces(P \circ Q)$ and $traces(P\$Z) = traces(P)$ by definitions.

$$(2)\quad ((a \to P) \circ (a \to Q))\$\Sigma! =_F a \to (P \circ Q)\$\Sigma!$$

Proof: $(a \to P) \circ (a \to Q) =_F a \to (P \circ Q)$ by Teorem 1(1).

$$(3.1)\quad ((a \to P) \circ (b \to Q))\$\Sigma! =_F (a \to P\$\Sigma!) \sqcap (b \to Q\$\Sigma!)$$

where $a \in \Sigma!, b \in \Sigma!$.
Proof:

$$\begin{aligned} &((a \to P) \circ (b \to Q))\$\Sigma! \\ &\quad =_F ((a \to P) \Box (b \to Q))\$\Sigma! \\ &\quad =_F (a \to P\$\Sigma!) \sqcap (b \to Q\$\Sigma!) \end{aligned}$$

First transform is derived by Theorem 1(2) and second transform is derived by Theorem 2(2).

$$(3.2)\quad ((a \to P) \circ (b \to Q))\$\Sigma! =_F (a \to P\$\Sigma!) \Box (b \to Q\$\Sigma!)$$

where $a \notin \Sigma!, b \notin \Sigma!$.
Proof:

$$\begin{aligned} &((a \to P) \circ (b \to Q))\$\Sigma! \\ &\quad =_F ((a \to P) \Box (b \to Q))\$\Sigma! \\ &\quad =_F (a \to P\$\Sigma!) \Box (b \to Q\$\Sigma!) \end{aligned}$$

First transform is derived by Theorem 1(2) and second transform is derived by Theorem 2(1).

In these proofs, we can apply theorems to sub-formulas because our operators preserves the failures-refinement.

Using $\circ$ operator, we can compute $\mathcal{P}^{st}_D(C, S)$ as follows.

$$\mathcal{P}^{st}_D(C, S) = (\bigcirc_{T \in \mathcal{T}_D(C,S)} @ \mathcal{P}^{seq}(T))$$

where

$$\mathcal{T}_D(C, S) = \{\ T \in tr(D) \mid C = comp(T), S = prev(T)\}$$

and $tr(D)$ is the set of all transitions in diagram $D$, $comp(T)$ is the component for transition $T$, and $prev(T)$ is the previous state for transition $T$.

$$\begin{aligned} &tr(D) = \{T \mid \exists SD.\ T \in SD, SD \in D\} \\ &comp(T) \in \{C \mid \exists S1.\exists M.\exists S2.\ T = (C, S1, M, S2)\} \\ &prev(T) \in \{S1 \mid \exists C.\exists M.\exists S2.\ T = (C, S1, M, S2)\} \end{aligned}$$

From the definitions, following property holds.

$$\sqcap_{\ T \in \mathcal{T}_D(C,S)} @ \mathcal{P}^{seq}_D(T) \sqsubseteq_F \mathcal{P}^{st}_D(C, S)$$

For example, the System component in Fig. 1 behaves as follows.

$$\begin{aligned} \mathcal{P}^{st}_{D_1}(System, d) &= \mathcal{M}(Login) \to \mathcal{P}^{st}_{D_1}(System, i_3) \\ &\quad \circ \mathcal{M}(Add) \to \mathcal{P}^{st}_{D_1}(System, i_4) \\ &\quad \circ \mathcal{M}(Login) \to \mathcal{P}^{st}_{D_1}(User, i_6) \\ \mathcal{P}^{st}_{D_1}(System, i_3) &= \mathcal{M}(OK) \to \mathcal{P}^{st}_{D_1}(System, d) \\ \mathcal{P}^{st}_{D_1}(System, i_4) &= \mathcal{M}(OK) \to \mathcal{P}^{st}_{D_1}(System, d) \\ \mathcal{P}^{st}_{D_1}(System, i_6) &= \mathcal{M}(NG) \to \mathcal{P}^{st}_{D_1}(System, d) \end{aligned}$$

If the initial state for each component $Init$ is given, we can define a CSP process representing whole design.

$$\begin{aligned} &\mathcal{P}^{design}_D(Init) \\ &\quad = \|_{C \in C_D} @ [\Sigma_D(C)]\, \mathcal{P}^{st}_D(C, Init(C))\$\Sigma!_D(C) \end{aligned}$$

where

$$\begin{aligned} &C_D = \{C \mid \exists T \in tr(D).\ C = comp(T)\} \\ &\Sigma_D(C) = \{\mathcal{M}(M) \mid \exists T \in tr(D).\ M = mes(T), C = comp(T)\} \\ &\Sigma!_D(C) = \{\mathcal{M}(M) \mid \exists T \in tr(D).\ M = mes(T), C = sender(M)\} \end{aligned}$$

and $mes(T)$ is the message for transition $T$ and $sender(M)$ is the sender for message $M$.

$$\begin{aligned} &mes(T) \in \{M \mid \exists C.\exists S1.\exists S2.\ T = (C, S1, M, S2)\} \\ &sender(M) \in \{CS \mid \exists MN.\exists CR.\ M = (MN, CS, CR)\} \end{aligned}$$

$\|_i @ [X_i]\, P_i$ is a parallel composition of all processes $P_i$ with events $X_i$.

$$\|_{i \le n} @ [Xi]\, Pi = ((P_0\,{}_{X_0}\|_{X_1} P_1)\,{}_{X_0 \cup X_1}\|_{X_2} P_2) \cdots {}_{(X_0 \cup \cdots \cup X_{n-1})}\|_{X_n} P_n$$

For example, the user and the system in Fig. 1 behave as follows.

$$\begin{aligned} &\mathcal{P}^{st}_{D_1}(User, d)\$\{\mathcal{M}(Login), \mathcal{M}(Add)\} \\ &\quad {}_{\Sigma_{D_1}}\|_{\Sigma_{D_1}} \mathcal{P}^{st}_{D_1}(System, d)\$\{\mathcal{M}(OK), \mathcal{M}(NG)\} \end{aligned}$$

where $\Sigma_{D_1} = \{\mathcal{M}(Login), \mathcal{M}(Add), \mathcal{M}(OK), \mathcal{M}(NG)\}$. It represents that the user starts from the default state and its sending message is *Login* and *Add*, and the system starts from the default state and its sending message is *OK* and *NG*.

When the sequence diagram is refined with component dividing, the sending and receiving between components that have been defined before refinement are preserved. Therefore, the correctness of the refinement can be verified with equality of CSP processes.

$$\begin{aligned} &\mathcal{P}^{design}_D(Init) \setminus H(D, D') \\ &\quad =_F \mathcal{P}^{design}_{D'}(Init') \setminus H(D, D') \end{aligned}$$

where $D'$ is a design refined from $D$, $H(D, D')$ is a difference between $D$ and $D'$.

$$H(D, D') = (\Sigma_D \cup \Sigma_{D'}) - (\Sigma_D \cap \Sigma_{D'})$$

where "−" means the difference of the sets, and $\Sigma_D = \cup_{C \in C_D}(\Sigma_D(C))$.

In sequence diagrams, developers may or may not write concrete data on messages. Without concrete data, messages are nondeterministically selected by sender processes and deterministically received by receiver processes. While other formal methods like I/O automata can express concrete data passing, CSP failures model can verify models without concrete data.

### 4.3 Transformation to Standard CSP Process

We present a method for transforming any process $\mathcal{P}_D^{st}(C, S)\$\Sigma!$ to a failure-equivalent process $Synth_D(C, \mathcal{S})$ in standard CSP.

$$Synth_D(C, \mathcal{S}) =_F (\bigcirc_{S \in \mathcal{S}} @\mathcal{P}_D^{st}(C, S))\$\Sigma!_D(C)$$

where $\Sigma!_D(C)$ is the set defined in Sect. 4.2 and it means that the tools for standard CSP can be applied to $Synth_D(C, \mathcal{S})$ in order to verify the extended CSP processes with $\circ$ and \$ for sequence diagrams

Synth is defined as follows:

(1) $(A!_D(C, \mathcal{S}) = \phi) \Rightarrow$
$Synth_D(C, \mathcal{S}) =$
$\Box_{a \in A?_D(C,\mathcal{S})} @a \rightarrow Synth_D(C, N_D(C, \mathcal{S}, a))$

(2) $(A!_D(C, \mathcal{S}) \neq \phi \wedge A?_D(C, \mathcal{S}) = \phi) \Rightarrow$
$Synth_D(C, \mathcal{S}) =$
$\sqcap_{a \in A!_D(C,\mathcal{S})} @a \rightarrow Synth_D(C, N_D(C, \mathcal{S}, a))$

(3) $(A!_D(C, \mathcal{S}) \neq \phi \wedge A?_D(C, \mathcal{S}) \neq \phi) \Rightarrow$
$Synth_D(C, \mathcal{S}) =$
$\sqcap_{a \in A!_D(C,\mathcal{S})} @a \rightarrow Synth_D(C, N_D(C, \mathcal{S}, a))$
$\triangleright \Box_{a \in A?_D(C,\mathcal{S})} @a \rightarrow Synth_D(C, N_D(C, \mathcal{S}, a))$

where

$$N_D(C, \mathcal{S}, a) = \{next(T) \mid T \in tr(D),$$
$$C = comp(T), prev(T) \in \mathcal{S}, a = \mathcal{M}(mes(T))\}$$
$$A!_D(C, \mathcal{S}) = \{\mathcal{M}(M) \mid \exists T \in tr(D).$$
$$M = mes(T), C = sender(M), prev(T) \in \mathcal{S}\}$$
$$A?_D(C, \mathcal{S}) = \{\mathcal{M}(M) \mid \exists T \in tr(D).$$
$$M = mes(T), C = receiver(M), prev(T) \in \mathcal{S}\}$$

and $receiver(M)$ is the receiver for message $M$.

$$receiver(M) \in \{CR \mid \exists MN.\exists CS. \; M = (MN, CS, CR)\}$$

For example, the System component in Fig. 1 $\mathcal{P}_{D_1}^{st}(System, d)\$\{\mathcal{M}(M_3), \mathcal{M}(M_4)\}$ is transformed to the following process $Synth_{D_1}(System, \{d\})$.

$$Synth_{D_1}(System, \{d\})$$
$$= \mathcal{M}(Login) \rightarrow Synth_{D_1}(System, \{i_3, i_6\})$$
$$\Box\, \mathcal{M}(Add) \rightarrow Synth_{D_1}(User, \{i_4\})$$
$$Synth_{D_1}(System, \{i_3, i_6\})$$
$$= \mathcal{M}(OK) \rightarrow Synth_{D_1}(System, \{d\})$$
$$\sqcap\, \mathcal{M}(NG) \rightarrow Synth_{D_1}(System, \{d\})$$
$$Synth_{D_1}(System, i4)$$
$$= \mathcal{M}(OK) \rightarrow Synth_{D_1}(System, \{d\})$$

$\mathcal{S}$ is a subset of the component's state. So, in the worst case, the computational complexity to generate standard CSP process from a sequence diagram design is $O(2^N)$ for state count $N$. However, for practical cases, most of them are not reachable from the initial state. We can start calculation from the initial state and proceed with necessary states. For example, the system in Fig. 1 has 4 states ($2^N = 16$), but only 3 subsets of these are reachable from the default state. The translation finishes in a couple of seconds in our experiments using a laptop computer with 1.8 GHz CPU.

## 5. SD2CSP: A Conversion Tool

To find mistakes of sequence diagrams using CSP-tools such as the model checker FDR, a tool which converts sequence diagrams into CSP processes is required. We developed a tool named SD2CSP that converts the sequence diagrams into extended CSP processes with the synthesis operators $\circ$ and \$, and then transforms them to Standard CSP processes as explained in Sect. 4.3.

SD2CSP reads the standard XMI file where information in sequence diagrams is described, and it outputs the FDR description which contains the CSP processes of the sequence diagrams. The XMI format is a XML format provided as OMG standard to exchange the UML model. To read the XMI file by SD2CSP, the model elements must be lined up in the time series. The model checker FDR is a standard CSP-tool. Therefore, SD2CSP with help of FDR can verify refinements between sequence diagrams.

Figure 3 is a screen shot of SD2CSP. SD2CSP runs on Eclipse platform. Using UML editor plugins, sequence diagrams can be designed and converted to CSP processes seamlessly. Figure 4 is an example output for

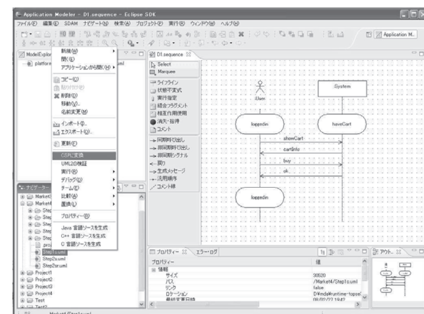

**Fig. 3** A screen shot of SD2CSP.

```
COMPO_(System.default_) = (
     login -> COMPO_(System.i5_i4_)
  [] addToCart -> COMPO_(System.i6_)
)

COMPO_(System.i5_i4_) = (
      ng -> COMPO_(System.default_)
  |~| ok -> COMPO_(System.default_)
)

COMPO_(System.i6_) = (
      ok -> COMPO_(System.default_)
)
```

**Fig. 4** A generated FDR description.

$Synth_{D_1}(Sytem, \{d\})$.

## 6. An Example of Verification

We demonstrate how SD2CSP with help of FDR can verify concurrent systems by a shopping site example. This shopping site provides following functions: displaying all sold items, adding items to the cart, purchasing, displaying recipe of recommended dish. Login and logout is necessary to use these functions.

In the abstract design level, we assume that all functions are achieved by a single component named System. In this example, state invariants are used for representing the login state and the logout state of the user, and the empty state and the non-empty state of the cart in the system. This abstract design level is modeled in 14 sequence diagrams (Fig. 5).

In the concrete design level, we classifies the functions of this shopping site under three features: users management, commodities selling, and recipe offering. Therefore, the component System is divided into the three components UserManager, MarketSite, and RecipeSite for the three functions (Fig. 6),

Now, by our tool SD2CSP with help of FDR, we can check whether the concrete sequence diagrams correctly refines the abstract sequence diagrams. As the result of this verification, we found an error. This error is analyzed and corrected as follows:

1. Analysis: According to the FDR debugger, the error occurs after the user logged out and logged in again. After the re-login, as a reply of the message “showCart”, system returns “empty” at the abstract level but “cartInfo” at the concrete level.
2. Conjecture: The component MarketSite sends the messages “empty” in the state “default_” and sends the message “cartInfo” in the state “haveCart”. Therefore, we can guess that the error is caused by forgetting updates of state of MarketSite.
3. Check: By checking the two right sequence diagrams in Figs. 5 and 6, we can find out the cause as follows. In the abstract level, if the message “logout” is sent, the system clears the cart. However, in the concrete level, MarketSite manages the cart, but nothing is sent to MarketSite.
4. Correction: Considering the information above, this error can be corrected by clearing the cart in the concrete level at each logout (i.e. the right one of Fig. 6). We add the message “clearCart” from System to MarketSite as shown in Fig. 7.

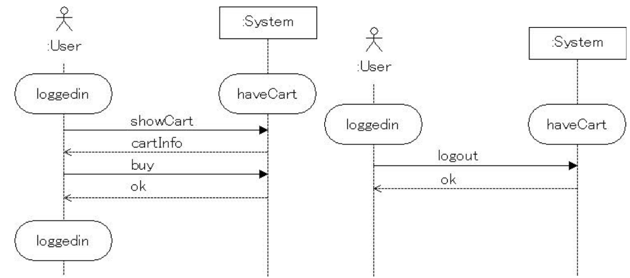


**Fig. 5** A part of sequence diagrams in the abstract level.

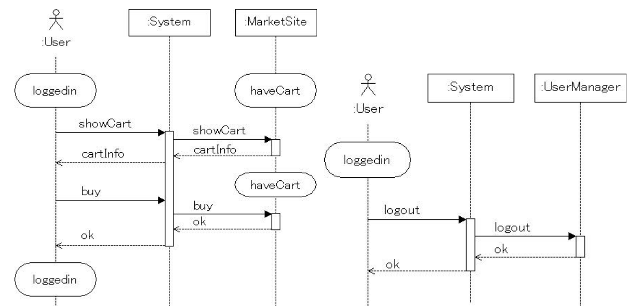


**Fig. 6** A part of sequence diagrams in the concrete level.

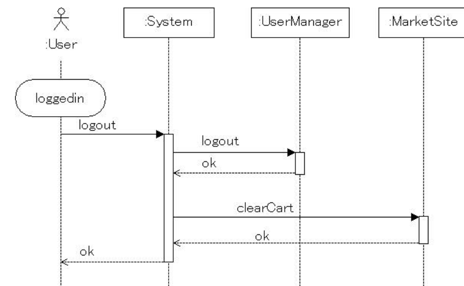


**Fig. 7** The fixed sequence diagram.

Then, we have confirmed that the corrected concrete sequence diagrams refines the abstract sequence diagrams by SD2CSP and FDR.

As shown in this example, using SD2CSP and FDR, we can check the refinement between two specifications at different abstract levels in the early stage. If an error is found in the refinement, we can correct the detected part. This tool helps improving the software quality without rework cost.

## 7. Related Works

There are some other studies for synthesis of state-based

model from scenario-based model.

A.W. Biermann has proposed a method for automatic program generation from execution step of sample calculation [6]. This method constructs state transition model from executed instructions and conditions satisfied in each step. Steps could be combined to one state if they have the same instruction. To generate deterministic program, states are divided if they have two or more transitions with the same condition and different destinations. The generation tool outputs a deterministic program with the minimum number of states.

E. Mäkinen has proposed a tool named "MAS" [7], in which a similar idea to [6] is applied to sequence diagrams. In this method, sending events are assigned to states and receiving events are used as transition conditions. MAS generates deterministic state transition model by a similar algorithm to [6]. When the generated model is different from the expectations, the model can be modified by entering counterexamples.

Biermann's method [6] and Makinen's method [7] generates compact deterministic state-based models that can execute input scenarios. However, these methods are not necessarily available in the early design stage because nondeterminism exists there. Our tool SD2CSP is available to sequence diagrams even if details in the components have not been decided because our method uses the failures model in CSP, where nondeterminism can be considered.

The approach using Live Sequence Chart has been proposed by D. Harel et al. [8]. Live Sequence Chart is an extension of sequence diagrams, it can describe a conditional scenario that must occur when the condition is satisfied. By explicitly giving such conditions in Live Sequence Chart, the meaning of the scenarios are clarified, the state transition model satisfying the conditions can be derived.

However, in the actual software development, to exactly define all the conditions of scenarios is not easy because the conditions can depend on internal states in objects which have not been fixed in the early design stage. Our method is available for normal sequence diagrams without such explicit conditions.

R. Alur et al. have proposed a method using a chart named "HMSC" [9]. HMSC shows the execution order between message sequence charts. Based on this approach, Message Sequence Chart plugin (LTSA-MSC), which is a plugin for LTSA model checker, has been developed [10]. LTSA-MSC verifies message sequence charts and a HMSC on LTSA model checker.

When using LTSA-MSC, nondeterminism can be described in HMSC, but behaviors described in each sequence chart must be deterministic. Therefore, if sequence charts contain nondeterminism, they must be carefully divided at the nondeterministic points to create the HMSC model. Each component selects a message it sends, but the selection procedure may not be fixed in an abstract level at the early stage. In our approach, such selection can be nondeterministically expressed by the internal choice $\sqcap$ in CSP.

## 8. Conclusion

In this paper, we have extended the process algebra CSP with two new operators $\circ$ and \$ for formally describing behaviors of components in sequence diagrams and for verifying them. Furthermore, we have proved CSP laws for $\circ$ and \$ as shown in Theorems 1 and 2, and we have presented a method for transforming processes which contain $\circ$ and \$ into standard processes, according to the CSP laws. The method allows us to use standard CSP-tools such as FDR.

We have developed a tool SD2CSP which is an implementation of our method, thus it generates CSP processes with $\circ$ and \$ from sequence diagrams and then transforms them to standard CSP processes in FDR syntax. SD2CSP can detect errors by verifying whether the concrete sequence diagrams behaves like the abstract sequence diagrams. We have demonstrated a sequence of usage of SD2CSP by an example of a shopping site.

It is a future work to extend SD2CSP with data-passing. Currently, data-passing between components has not been supported in SD2CSP yet, but various data-types such as integer, list, set, and user-defined type are supported in FDR. Therefore we believe it is feasible to verify data-passing. Also, we are discussing the refinements of data-types.

And in the future, we plan to improve the usability of our tool SD2CSP, by extending preprocesses of sequence diagrams. We plan following preprocesses:

- Automatic insertion of state invariants from additional scenarios
- Automatic complement of abnormal sequence diagrams

## Acknowledgments

We wish to express our gratitude to Professor Shinichi Honiden of NII and Professor Koichiro Ochimizu of JAIST for giving us the opportunity of our cooperative work. This work was partially supported by KAKENHI 20500023.

### References

[1] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[2] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.
[3] Formal Systems (Europe) Limited, "FDR2 User Manual," http://www.fsel.com/fdr2_manual.html.
[4] B. Buth and M. Schronen, "Model-checking the architectural design of a fail-safe communication system for railway interlocking systems," J.M. Wing, J. Woodcock, and J. Davies, ed., FM99, LNCS 1709, pp.1869–1869, Springer, 1999.
[5] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe, The modelling and analysis of security protocols: the csp approach, Addison-Wesley, 2001.
[6] A.W. Biermann and R. Krishnaswamy, "Constructing programs from example computations," IEEE Trans. Softw. Eng., vol.SE-2,

no.3, pp.141–153, 1976.
[7] E. Mäkinen and T. Systä, "Mas — An interactive synthesizer to support behavioral modelling in uml," Proc. 23rd International Conference on Software Engineering, 2001.
[8] D. Harel and H. Kugler, "Synthesizing state-based object systems from lsc specifications," Int. J. Foundations of Computer Science, 2002.
[9] R. Alur and M. Yannakakis, "Model checking of message sequence charts," Proc. 10th International Conference on Concurrency Theory, 1999.
[10] S. Uchitel, R. Chatley, J. Kramer, and J. Magee, "Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios," Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2003.
[11] H. Liang, J. Dingel, and Z. Diskin, "A comparative survey of scenario-based to state-based model synthesis approaches," Proc. 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, 2006.

**Tomohiro Kaizu** received his B.Eng. and M.Eng. degree in Department of Computer Science of Tokyo Institute of Technology in 2004 and 2006 respectively. In 2006–2010, he worked for NEC Corporation. In 2010, he joined Google Inc. He is currently under the Doctoral program in Information Science at Japan Advanced Institute of Science and Technology. His research interests include formal verification of software designs.

**Yoshinao Isobe** received his B.Eng. and M.Eng. degrees in Electrical Engineering from Shibaura Institute of Technology in 1990 and 1992 respectively. In 1992, he joined Electrotechnical Laboratory, MITI. He received his D.Eng. degree from Shizuoka University in 2001. He was a visiting researcher of the University of Wales, Swansea for one year in 2003. He is currently a senior researcher in the National Institute of Advanced Industrial Science and Technology and a special appointment associate professor in the National Institute of Informatics. His research interests include formal verification of concurrent systems. He is a member of JSSST and IPSJ.

**Masato Suzuki** received his B.Eng. degree in Department of Computer Science of Tokyo Institute of Technology (TITECH) in 1987, and his M.Eng. and D.Eng. degrees from Graduate School of TITECH in 1989 and 1992 respectively. He was a research associate of the Japan Advanced Institute of Science and Technology (JAIST), moved TITECH as an associate professor in 1998. In 2002–2004, he was also an associate professor of National Institute of Informatics. He is currently an associate professor of School of Information Science, JAIST. His research interests include architecture and component based software engineerings. He is a member of JSSST and IPSJ.