1. Introduction

In recent years, the internationalization of activities and information technology in companies has intensified competition among them. Companies are under pressure to quickly respond to business needs, and the period for making changes to existing business and launching new businesses has been shortened. Therefore, the need to quickly change or build information systems has been increasing. Under such circumstances, service-oriented architecture (SOA) [3] has been attracting attention as the architecture of information systems. In SOA, an information system is built by composing independent software units called services.

In this paper, we consider the problem of synthesizing a concrete model from an abstract specification, where the concrete model describes the detailed behavior of services. We assume that the behavior of services is written by state machines; the abstract specification describes how services interact with each other. It is not easy for designers to design a concrete model directly from requirements because these requirements contain huge gaps. However, defining an abstract specification is relatively simple. Therefore, if we can automatically synthesize a concrete model from a high-quality abstract specification, the designer’s workload would significantly reduce and product quality would improve.

In the field of software engineering, several investigations synthesize a concrete model from an abstract specification. Harel et al. proposed a methodology for synthesizing statechart models from scenario-based requirements [4]. Whittle et al. proposed a methodology for synthesizing hierarchical state machine models from expressive scenario descriptions [5]. Liang et al. defined a set of comparison criteria and surveyed 21 different synthesis approaches presented in the literature based on comparison criteria [6].

In SOA, the problem of synthesizing a concrete model from an abstract specification is known as the choreography realization problem (CRP) [7], [8]. The abstract specification, called choreography, is defined as a set of interactions among services, which are given by a dependency relation between sent and received messages; the concrete model is called service implementation, which defines the behavior of the service. This paper uses the communication diagram and the state machine of Unified Modeling Language (UML) 2.x [9] to describe the choreography and service implementation, respectively.

Bultan and Fu formally study CRP [8]. They used collaboration diagrams of UML 1.x and showed that the conditions for the given choreography are realizable. In addition, they showed a method for representing service implementation as the state space in which a state is defined as a set of unsent messages, and another method for synthesizing a set of finite state machines with projection mapping. However, the synthesized state machines are not intelligible because the number of states increases exponentially as the number of messages increases. Furthermore, they adopt the semantics that message send and receive events for a synchronous call occur sequentially. Under these semantics, the UML specification that “the execution of the call operation action waits until the execution of the invoked behavior completes and a reply transmission is returned to the caller” [9] cannot be represented.

Miyamoto et al. proposed a method for synthesizing hierarchical state machines from the choreography given in communication diagrams [10]. In this method, dependency constraints between message send and receive events are represented by Petri nets [11]; state machines are synthesized from their reachability space. A method for extracting
the hierarchical structure by analyzing the reachability space has been provided; however, this technique can be applied only to simple cases.

Intelligibility, however, is highly subjective and it is difficult to discuss this concept quantitatively. Cruz-Lemus et al. experimentally evaluated the relationship between some metrics of state machines and the time taken to understand them [12]. According to the results, state machines are more easily understood as values of following metrics become small: the number of simple states (NSS), number of transitions (NT), number of guards (NG), and number of do-activities (NA). Because we do not use do-activity in this paper, we use the first three metrics (NSS, NT, and NG) for intelligibility.

In this paper, we propose a method for converting a Petri net directly into a state machine. It is shown that there is a relationship between the possibility of direct conversion and the structural properties of Petri nets. Initially, the proposed method converts the Petri net to satisfy the structural properties and subsequently, it converts Petri nets into hierarchical state machines without generating their state spaces.

The remainder of this paper is organized as follows. Sect. 2 introduces a UML subset, called the subset of UML for formally describing choreography and behavioral feature (cbUML), to discuss CRP and an extended Petri net, called the message mark graph (MMG). Sect. 3 describes CRP and the proposed method, called the Construct State-machine Cutting Bridges (CSCB) method. Sect. 4 evaluates the CSCB method in terms of intelligibility. Note that in this paper, it is assumed that the choreography is presented in a single communication diagram as the first stage of the study. Sect. 5 presents the conclusion.

2. Preliminaries

2.1 cbUML

Let us introduce a subset of UML called cbUML. The complete set of cbUML is described in [13]. This section shows a simplified version of cbUML, which is sufficient for the discussion of this paper.

Definition 1 (cbUML): A cbUML model is a tuple \( (C, M, \mathcal{A}, CD, SM) \), where \( C \) is the set of classes, \( M \) is the set of messages, \( \mathcal{A} \) is the set of attributes, \( CD \) is the set of communication diagrams, and \( SM \) is the set of state machines.

One class exists for each service and a state machine defines its behavior. A communication diagram describes a scenario, which is an interaction of services. In this paper, we assume that \( |CD| = 1 \). Each message and attribute is owned by a class and a message corresponds to the method of the class that receives the message.

2.1.1 Messages

The set \( M \) of messages is partitioned by the type of messages: \( M = M_{\text{top}} \cup M_{\text{top}} \cup M_{\text{rep}} \), where \( M_{\text{top}} \) is the set of synchronous messages generated by synchronous calls, \( M_{\text{top}} \) is the set of asynchronous messages generated by asynchronous calls, and \( M_{\text{rep}} \) is the set of reply messages to synchronous messages. Let \( M_{s} = M_{\text{top}} \) and \( M_{a} = M_{\text{top}} \cup M_{\text{rep}} \). Correspondence between the synchronous call and its reply is given by the function \( \text{ref} : M \rightarrow M \cup \{\text{nil}\} \), such that \( \forall m \in M_{\text{top}} : \text{ref}(m) \in M_{\text{rep}} \), \( \forall m \in M_{\text{rep}} : \text{ref}(m) \in M_{\text{top}} \), \( \forall m \in M_{\text{top}} : \text{ref}(m) = \text{nil} \), and \( \forall m \in M_{\text{top}} \cup M_{\text{rep}} : \text{ref}(\text{ref}(m)) = m \).

The services behave differently during interactions depending on the type of message as follows. In the case of a synchronous call, the caller’s execution is suspended until the caller receives a reply from the callee. However, in the case of an asynchronous call, the caller can continue to operate, regardless of the behavior of the callee.

In UML, each message has two events: a send event and a receive event. For a synchronous message, the receive event occurs immediately after the send event. However, for a discussion that occurs subsequently, we need two events that occur sequentially. Therefore, we define that each synchronous message has two events: a preparation event for message sending and a send-receive event where the preparation event is a caller’s event and the send-receive event is a callee’s event. The preparation event and the send-receive event of a synchronous message \( m \in M_{s} \) are denoted by \( \text{?}m \) and \( \text{!}m \), respectively. For an asynchronous or a reply message \( m \in M_{a} \), the send and receive events are denoted by \( \text{?}m \) and \( \text{!}m \), respectively. Hereafter, an active event is the send-receive event of a synchronous message or the send event of an asynchronous or a reply message. The set \( \Sigma \) of message events

\[ \Sigma = \{\text{?}m, \text{!}m \mid m \in M_{s}\} \cup \{\text{?}m, \text{!}m \mid m \in M_{a}\}, \]

and \( \Delta \) of active events are defined as follows:

\[ \Delta = \{\text{!}m \mid m \in M\}. \]

2.1.2 Communication Diagrams

Communication diagrams explain interactions in which the arcs between the communicating lifelines are decorated with message descriptions.

Definition 2 (Communication Diagram): A communication diagram \( cd \in CD \) is a tuple \( cd = (C^{cd}, M^{cd}, Conncd, line^{cd}, D^{cd}) \), where \( C^{cd} \subseteq C \) is the set of class, which are called lifelines and correspond to services; \( M^{cd} \subseteq M \) is the set of messages; \( Conncd \subseteq C^{cd} \times C^{cd} \) is the set of connectors, which is given as a symmetric relation on \( C^{cd} \); \( line^{cd} : M^{cd} \rightarrow Conncd \) assigns a connector for each message; and \( D^{cd} \subseteq \Delta \times \Delta \) indicates a dependency relation among active events, where \( D^{cd} \) must be acyclic.

Superscripts may be omitted if the context is clear.

In UML communication diagrams, sequence number is provided to define an order of messages. However, the number is not sufficient to describe complex behavior. We
use dependency relation $D^{rd}$ as the alternative to sequence number.

A conversation is a sequence of messages exchanged among services [8]. The set of conversations defined by a communication diagram $cd$ is denoted by $\mathcal{C}(cd) \subseteq \mathcal{M}^r$, where $\mathcal{M}^r$ is the set of all sequences of distinct messages.

**Definition 3:** A conversation $\sigma = m_1m_2 \cdots m_n$ is in $\mathcal{C}(cd)$ if and only if $\sigma \in \mathcal{M}^r$ and the corresponding sequence $\gamma = m_1\gamma_2 \cdots \gamma_n$ of active events satisfy $\forall i, j \in [1..n]: (\gamma_i, \gamma_j) \in D \Rightarrow i < j$.

Figure 1 shows a communication diagram. In this example, messages Req1 and Check5 are synchronous messages and the dashed lines with open arrow heads are the reply messages. Assume that the dependency relation among active events is given as shown in Fig. 2, where rhombuses, rectangles, and ellipses indicate synchronous calls, their reply, and asynchronous calls, respectively. The following sequence is a conversation of this example.

$$\sigma = \text{Req1 Check1 Req1_rep Check2 Check3}$$

2.1.3 State Machines

The behavior of each service is described by a state machine.

**Definition 4** (State Machine): A state machine is a tuple $sm = (V, R, r^t, \Theta, \Phi, E, C, B)$, where $V$ is the set of vertices, $R$ is the set of regions, $r^t \in R$ is the top region, $\Theta$ is an ownership relation between vertices and regions, $\Phi$ is the set of transitions, $E$ is the set of events, $C$ is the set of constraints, and $B$ is the set of behaviors.

In UML state machines, although there are various kinds of states and pseudo-states, only simple states, composite states, final states, and initial pseudo-states are used in this paper. Therefore, the set $V$ of vertices is partitioned into the following types of subsets: $V = SS \cup CS \cup FS \cup IS$, where $SS$ is the set of simple states, $CS$ is the set of composite states, $FS$ is the set of final states, and $IS$ is the set of initial pseudo-states.

A region, except for the top region, is owned by a composite state and a composite state is owned by a region. The ownership relation $\Theta$ is defined as a function from $(V \cup R) \setminus \{r^t\}$ to $(CS \cup R)$ and $\Theta(x_1) = x_2$ means that $x_1$ is owned by $x_2$. For $x \in V \cup R$, let $des(x) = \{x' \mid \exists i > 0 : \Theta(x') = x\}$ be the set of descendants of $x$, where $\Theta^i(x) = \Theta(\Theta(x)) (i > 1)$. The top region $r^t$ exists in the root of each state machine; this region is not owned by any composite state, and every state and region in any composite state are descendants of the top region.

**Definition 5** (Orthogonal State): Two vertices $v_1, v_2 \in V$ are called orthogonal and denoted by $v_1 \perp v_2$ if there exist different regions $r_1, r_2 \in R$ such that $r_1 \neq r_2, \Theta(r_1) = \Theta(r_2)$, $v_1 \in des(r_1)$, and $v_2 \in des(r_2)$.

**Definition 6** (Consistent State): A set $\hat{V} \subset V$ of vertices is called consistent if and only if for each $v_1, v_2 \in \hat{V}$, if $v_1 \neq v_2$ then $v_1 \perp v_2$, $v_1 \in des(v_2)$, or $v_2 \in des(v_1)$.

The set $E$ of events is given as $E = \Sigma \cup \{\tau\}$, where $\Sigma$ is the set of message events in the state machine and $\tau$ is the completion event that occurs when a transition with no trigger event fires.

A transition $tr \in \Phi$ is a tuple $tr = (src, tri, grd, eff, tgt)$, where $src \in V$ is the originating vertex of the transition, trigger $tri \in E$ is the event that makes the transition fire, guard $grd \in C$ is a condition to fire, effect $eff \in B$ is an optional behavior to be performed when the transition fires, and $tgt \in V$ is the target vertex. The set $\{src, tgt\}$ must not be consistent. A caller’s message sending event becomes an effect and a callee’s message receiving event becomes a trigger; therefore, $\Sigma \subseteq B$. The set $B$ of behaviors may contain an effect that manipulates the attributes of the corresponding class. A guard condition must be a Boolean expression and the attributes of the corresponding class may be used. According to the UML specification [9], triggers, guards, and effects are denoted as “$\text{tri}[\text{grd}] / \text{eff}$” in diagrams.

Owing to space limitations, the details of the operational semantics of state machines are omitted. They have been developed based on [14], [15], and reported in [13]. A state machine has a message pool and its state is defined by a consistent set of active states, set of suspended regions, set of messages in the message pool, and values of the attributes. A transition may fire when the originating vertex is active, the message of the trigger event is in the message pool or it is the completion event, and the guard is true. When the transition fires, the originating vertex and its descendants are inactivated, the message is removed from the message pool, the effect is executed, and the target vertex and initial pseudo-states in the first descendant regions are activated. The steps for synchronous calls and asynchronous calls are explained with examples.

Figure 3 shows the execution steps of an asynchronous...
call. The gray states are active. When state machine \( sm1 \) transitions from state \( s11 \) to state \( s12 \) due to the completion event, an asynchronous call is executed. At this time, the send event \( !m \) occurs and message \( m \) is added to the message pool of \( sm2 \). The state machine \( sm2 \) transitions from state \( s21 \) to state \( s22 \), consuming message \( m \) due to the receive event \( ?m \).

Figure 4 shows the execution steps of a synchronous call. A synchronous call is executed in \( sm1 \). At this time, the preparation event \( $m \) occurs in \( sm1 \) and the region that contains the transition is suspended. Moreover, message \( m \) is added to the message pool of \( sm2 \). State machine \( sm2 \) transitions from state \( s21 \) to \( s22 \), consuming message \( m \) by the occurrence of the send-receive event \( !m \). Then, \( sm2 \) sends a reply message \( rm \) to \( sm1 \) on transitioning from \( s22 \) to \( s23 \). At this time, the send event \( !m \) occurs, and message \( rm \) is added to the message pool of \( sm1 \). Now, \( sm1 \) releases the suspended region and transitions from state \( s11 \) to state \( s12 \), consuming reply message \( rm \) by the occurrence of the receive event \( ?rm \). Note that the receive event \( ?rm \) does not appear in the state machine because we are using the region suspend mechanism.

The set of all conversations, obtained by the execution of a set \( SM \) of state machines, is denoted by \( C(SM) \).

2.2 Petri Nets

A Petri net [11] is a tuple \( N = (P, T, F) \), where \( P \) is the set of places, \( T \) is the set of transitions, and \( F \subseteq P \times T \cup T \times P \) is the flow relation. For \( x \in P \cup T \), the set \( \{ y \in P \cup T \mid (y, x) \in F \} \) is called the preset of \( x \) and denoted by \( \bullet x \). Similarly, the set \( \{ y \mid (x, y) \in F \} \) is called the postset of \( x \) and is denoted by \( x \bullet \). For set \( X \), \( \bullet X = \bigcup_{x \in X} \bullet x \) and \( x \bullet = \bigcup_{x \in X} x \bullet \).

A place \( p \in P \) is called a source place and a sink place when \( \bullet p = \emptyset \) and \( p \bullet = \emptyset \), respectively. Similarly, a transition \( t \in T \) is called a source transition and a sink transition when \( \bullet t = \emptyset \) and \( t \bullet = \emptyset \), respectively. A transition \( t \) is called a fork transition and a join transition when \( |\bullet t| > 1 \) and \( |t \bullet| > 1 \), respectively. The sets of join transitions and fork transitions are denoted by \( T_{join} \) and \( T_{fork} \), respectively. Under the standard definition of Petri nets, if \( \forall p \in P : |\bullet p| = 1 \) and \( |p \bullet| = 1 \), then the Petri net is called a marked graph. In this paper, we relax the condition as follows: \( \forall p \in P : |\bullet p| \leq 1 \) and \( |p \bullet| \leq 1 \).

Definition 7 (MMG): An MMG is a tuple \( N = (P, T, F, A) \), where the underlying Petri net \( (P,T,F) \) satisfies the following conditions:

1. \( N \) is acyclic,
2. Only one source place \( p_s \) and one sink place \( p_e \) exist,
3. No source transitions and sink transitions exist, and
4. \( |p_s \bullet| = 1, |p_e \bullet| = 1 \), and \( \forall p \in P \setminus \{p_s, p_e\} : |\bullet p| = 1, |p \bullet| = 1 \).

\( G : T \rightarrow 2^P \) is a firing constraint and the partial function \( A : T \rightarrow \Sigma \cup \{\epsilon\} \) assigns an event for each transition \(^1\).

A state of an MMG is expressed by a pair \( (M, X) \), where \( M : P \rightarrow \{0,1\} \) is a marking and \( X : T \rightarrow \{\text{true}, \text{false}\} \) is a firing configuration. The initial state \( (M_0, X_0) \) of an MMG is given as follows:

\[
M_0(p) = \begin{cases} 1 & \text{if } p = p_s \\ 0 & \text{otherwise, and} \end{cases}
X_0(t) = \text{false} \quad (\forall t \in T).
\]

A transition \( t \in T \) is enabled if and only if \( \forall p \in \bullet t : M(p) \geq W_t(p, t) \) and \( \forall t' \in G(t) : X(t') = \text{true} \), where \( W_t(x, y) = 1 \) if \( (x, y) \in F \) and \( W(x, y) = 0 \) otherwise. A new state \( (M', X') \), which is obtained by the firing of transition \( t \) and expressed by \( (M, X)(t)(M', X') \), is given as follows:

\[
M'(p) = M(p) - W_t(p, t) + W(t, p), \text{ and}
X'(t') = \begin{cases} \text{true} & \text{if } t = t' \\ X(t') & \text{otherwise.} \end{cases}
\]

A sequence of transitions \( \sigma = t_1 \cdots t_n \) is called a possible firing sequence from a state \( (M_0, X_0) \) if \( (M_0, X_0)(t_1)(M_1, X_1) \cdots (t_n)(M_n, X_n) \). The set of all possible firing sequences from state \( (M_0, X_0) \) in net \( N \) is denoted by \( LN(M_0, X_0) \). For sequence \( \sigma = t_1 \cdots t_n \), \( A(\sigma) = A(t_1) \cdots A(t_n) \).

Definition 8: Let \( PN \) be a set of MMGs. A communication composition of MMGs \( CC(PN) = (P, T, F, A) \) is defined as follows:

\[
\hat{P} = (\cup_{\forall p \in PN} P_n) \cup \{p_m \mid m \in M\},
\hat{T} = \cup_{\forall p \in PN} T_n, \hat{F} = \cup_{\forall p \in PN} F_n \cup \{(t_m, p_m), (p_m, t_n) \mid A(t) = \langle m, m \in M_n \rangle \cup \{(t_n, p_m), (p_m, t_n) \mid \hat{A}(t) = Sm, \hat{A}(t) = \langle m, m \in M_n \rangle, \hat{G} : \hat{T} \rightarrow 2^\hat{P} \text{ s.t. } \hat{G}(t) = G_n(t), t \in T_n, \text{ and } \hat{A} : \hat{T} \rightarrow \Sigma \cup \{\text{false}\} \rangle = A_n(t), t \in T_n \}. \text{ The initial state } (\hat{M}_0, \hat{X}_0) \text{ is extended in a similar manner.}
\]

\(^1A(t) = \epsilon \) means that no event is assigned to \( t \).
is similar to that of MMGs except that it has several source and sink places. The set \( \mathcal{C}(\mathcal{P}N) \) of all conversations of a set \( \mathcal{P}N \) of MMGs is defined using \( \mathcal{L}C^{(\mathcal{P}N)}(\hat{\mathcal{M}}_0, \hat{\mathcal{X}}_0) \).

Definition 9 (Bridge [16]): Let \( N = (P, T, F) \) and \( N_1 = (P_1, T_1, F_1) \) and \( N_2 = (P_2, T_2, F_2) \) be subnets of \( N \). An elementary path \( \pi = (n_1, \ldots, n_r) \), \( r \geq 2 \) is a bridge from \( N_1 \) to \( N_2 \) if and only if \( \pi \cap (P_1 \cup T_1) = \{n_1\} \) and \( \pi \cap (P_2 \cup T_2) = \{n_r\} \). When \( n_1 \) and \( n_r \) are transitions, the bridge is called a T-T bridge.

Definition 10 (Parallel Path): For two elementary paths \( \pi_1 = (n_1, \ldots, n_r) \) and \( \pi_2 = (n_1, \ldots, n_r) \), if \( \pi_1 \cap \pi_2 = \{n_1, n_r\} \), then \( \pi_1 \) and \( \pi_2 \) are called parallel paths.

For a transition \( t \in T \) in an MMG, \( FJ(t) \subseteq T \) (resp. \( JF(t) \subseteq T \)) is the set of terminal (resp. starting) transitions of parallel paths starting from (resp. terminating at) \( t \).

Definition 11 (Convertible MMG): An MMG is called a convertible MMG (CMMG) if the following conditions hold:

1. \( |T_{fork}| = |T_{join}| \).
2. \( T_{fork} \cap T_{join} = \emptyset \).
3. For any parallel paths \( \eta_1 \) and \( \eta_2 \), there exist no bridge from one to the other, and
4. If \( A(t) = \#m \), then \( t \bullet \bullet = \{t'\}, A(t') = \#ref(m) \).

### 3. Choreography Realization Problem

#### 3.1 Choreography Realization Problem

A single communication diagram describes a scenario, which is an interaction of services in the system. All the behaviors of the system are indicated by a set of communication diagrams; this is referred to as choreography.

Problem 1 (CRP): For a given set \( \mathcal{C}D \) of communication diagrams, is it possible to synthesize the set \( \mathcal{S}M \) of state machines that satisfy \( \mathcal{C}(\mathcal{C}D) = \mathcal{C}(\mathcal{S}M) \)? If possible, obtain the set of state machines.

In the case of un-realizable choreography, it is preferred that state machines that mimic the choreography as closely as possible are synthesized. A set of state machines that satisfy \( \mathcal{C}(\mathcal{C}D) \supseteq \mathcal{C}(\mathcal{S}M) \) is called a weak realization of the given choreography. Since a set of empty state machines, in which \( \mathcal{C}(\mathcal{S}M) = \emptyset \), is a weak realization for any choreography, the set of state machines whose \( \mathcal{C}(\mathcal{S}M) \) is maximal is expected.

Bultan et al. informally and formally introduced CRP in [7] and [8], respectively. There is a subtle difference between the definitions of weak realizability in [7] and [8]. The definition of weak realizability in [7] and in this paper is the same; the definition of weak realizability in [8] requires that \( \mathcal{C}(\mathcal{C}D) = \mathcal{C}(\mathcal{S}M) \); however, a state machine may get stuck without reaching a final state. When communication diagrams are inconsistent, it is not always possible to synthesize state machines that accept all conversations. Therefore, this paper adopts the definition in [7].

Bultan and Fu show sufficient realizability conditions for a class of collaboration diagrams, and a method for synthesizing a set of finite state machines by projection mapping [8]. They introduce the notion of well-informedness. In our formulation, well-informedness is represented as follows.

Definition 12: Given an event \( e \in \Sigma \) and the corresponding message \( m \) in a communication diagram \( cd = (C, M, Conn, line, D) \), the event \( e \) is called well-informed if for all \( e' \) and the corresponding message \( m' \) such that \( (e', e) \in D \), one of the following conditions hold: (1) \( e \) is a minimal event of \( D \), or (2) \( send(m) \in \{send(m'), recv(m')\} \), where \( send(m) \) (resp. \( recv(m) \)) shows the lifeline that sends (resp. receives) the message \( m \).

A communication diagram is called well-informed if all of its events are well-informed.

In the following subsections, we show a method for synthesizing intelligible state machines using the rich notation of UML state machines. Hereafter, we assume that the given \( CD \) is (weak) realizable, the set \( CD \) contains only one communication diagram, and the communication diagram is well-informed. This method does not assure maximality of synthesized state machines and this is a topic for future research.

#### 3.2 CSCB Method

The proposed CSCB method synthesizes state machines from a communication diagram.

1. Construct a precedence relation \( \Rightarrow \) on the set of events. For each service \( c \), perform the following steps.
2. Derive a precedence relation \( \Rightarrow' \) from \( \Rightarrow \).
3. Construct an MMG from \( \Rightarrow' \).
4. Cut T-T bridges from the MMG.
5. Separate the fork and join transitions in the MMG.
6. Find one-to-one correspondence between \( T_{fork} \) and \( T_{join} \) in the MMG.
7. Convert the CMMG into a state machine.

##### 3.2.1 Construction of precedence relation \( \Rightarrow \)

The precedence relation \( \Rightarrow \subseteq \Sigma \times \Sigma \) on the set of events is obtained by augmenting \( D \) as follows:

\[
\Rightarrow = D \cup \{(\#m, \#m) \mid m \in M_1 \} \cup \{(\#m, \#m) \mid m \in M_1 \} \\
\cup \{(\#m_1, \#m_2) \mid m_1 \in M_1, m_2 \in M_2, (m_1, m_2) \in D \} \\
\cup \{(\#m_1, \#m_2) \mid m_1 \in M_2, m_2 \in M_1, (m_1, m_2) \in D \} \\
\cup \{(\#m_1, \#m_2) \mid m_1 \in M_1, m_2 \in M_2, (m_1, m_2) \in D \}
\]

Figure 5 shows the precedence relation \( \Rightarrow \), which has been transitively reduced, for the communication diagram.
3.2.2 Deriving \(\Rightarrow^c\)

A relation \(Y_c\) for a service \(c\) is implicitly given (\(Y_c\) appears in both sides) as follows:

\[
Y_c = \{(e_1, e_2) \in \Sigma^* \times \Sigma^* | (e_1, e_2) \in \Rightarrow^+\} \cup \{(\text{init}, e), (e, \text{end}) | e \in \Sigma^*\} \cup \{(\text{?ref}(m), e) | m \in M_c, e \neq \text{?ref}(m), (\$m, e) \in Y_c\},
\]

where \(\Rightarrow^+\) is the transitive closure of \(\Rightarrow\). The first set is the projected relation on the set of events of service \(c\). The second set represents that the first dummy event “init” (resp. the last dummy event “end”) must precede (resp. succeed) all other events. The third set adds the additional constraints so that only the receive event “\(\text{?ref}(m)\)” of the reply message of a synchronous message \(m\) is the direct successor of the preparation event \(\$m\). Then, the precedence relation \(\Rightarrow^c\) for a service \(c\) is obtained by transitivity reducing \(Y_c\) as follows:

\[
\Rightarrow^c = Y_c^c.
\]

The precedence relations \(\Rightarrow^c\), which are derived from the precedence relation \(\Rightarrow\) shown in Fig. 5, are shown in Fig. 6. Here, because \((\$\text{Req1}, \text{?\text{Req1}}_\text{rep}), (\text{?\text{Req1}}, \text{?\text{Answer}}) \in \Rightarrow \text{Service1}\), a pair \((\text{?\text{Req1}}_\text{rep}, \text{?\text{Answer}})\) is added to \(\Rightarrow \text{Service1}\).

Because we assume that the realizability and additional constraints are added, the following lemma holds:

**Lemma 2:** \(\mathcal{C}(\Rightarrow) \supseteq \mathcal{C}(\mathcal{R})\), where \(\mathcal{R} = \bigcup_{e \in \mathcal{C}} \Rightarrow^c \cup \{(\$m, \$m) | m \in M_c\} \cup \{(\$m, \$m) | m \in M_d\}.

3.2.3 Constructing MMG

MMGs are constructed by converting the underlying set of relation \(\Rightarrow^c\) into transitions, adding a place for each pair in \(\Rightarrow^c\), and adding source and sink places. See Algorithm 1. Figure 7 (a) depicts the MMG of Service2.

Let \(\mathcal{P}N\) be the set of obtained MMGs. Then, the following lemma holds:

**Lemma 3:** \(\mathcal{C}(\mathcal{R}) = \mathcal{C}(\mathcal{P}N)\).

3.2.4 Cutting T-T bridges

In CMMGs (Def. 11, item 3.), bridges between two parallel paths should not exist; they are cut. Cutting all bridges, however, is not always necessary. Let \(U\) be a set of bridges and \(f : U \rightarrow 2^U\) be a function such that \(f(u)\) is a set of bridges that will not be bridges by cutting bridge \(u\). Then, finding the problem of setting the set of bridges results in the set shown in Figs. 1 and 2.

Let \(\mathcal{C}(\Rightarrow)\) be the set of conversations under the precedence relation \(\Rightarrow\). Then, the following lemma holds because \(\Rightarrow\) is obtained by just inserting events in \(\Sigma \setminus \Delta\).

**Lemma 1:** \(\mathcal{C}(C D) = \mathcal{C}(\Rightarrow)\).
After the set of bridges is found, Algorithm 2 is executed. Figure 7 shows an example, where (!Check1, p11, !Check2, p8, ?ReplyCheck2) of (a) is the only bridge. After edges (!Check1, p11) and (p8, ?ReplyCheck2) are removed, edges (Service2-init, p11) and (p8, Service2-end) are added. At that time, to avoid changing the behavior, the following firing conditions are added:

\[ G(t) = \begin{cases} 
!Check1 & \text{if } A(t) = !Check2 \\
!Check2 & \text{if } A(t) = ?\text{ReplyCheck2} 
\end{cases} \]

Because the behavior is preserved by using firing conditions, the set of conversations is also preserved.

3.2.5 Separating Fork and Join Transitions

If there exists a fork and join transition, it is split into a fork transition and a join transition, as shown in Fig. 8 (a).

3.2.6 Finding One-to-One Correspondence

Let A be a fork transition s.t. \(|FJ(A)| \geq 2\) and C ∈ FJ(A) be a join transition s.t. \(\exists t \in FJ(A) : t \Rightarrow C\). From the nonexistence of bridges, we can show that such a join transition exists uniquely. Let \(P_A \subseteq A\bullet\) be a set of output places of A that precede C: \(P_A = \{p \in A\bullet | p \Rightarrow C\}\). A dummy transition D is inserted between A and \(P_A\), as shown in Fig. 8 (b), where \(FJ(A) = \{B, C\}\) and \(P_A = \{E, F\}\). Then, a one-to-one correspondence exists between the pair C and D.

The above procedure is repeated from the maximal fork transition (in the sense of partial order), while a fork transition s.t. \(|FJ(\cdot)| \geq 2\) exists. Similar procedure is repeated for join transitions s.t. \(|JF(\cdot)| \geq 2\).

Because separating fork and join transitions and finding one-to-one correspondence do not affect behavior, the set of conversations is also preserved.

**Lemma 4:** The MMG obtained by applying steps 1–6 of the CSCB method is a CMMG.

**Proof:** The first condition of Definition 11 holds from step 6. The second condition holds from step 5. The third condition holds from step 4. The fourth condition holds from step 2, and steps 3 to 6 change the structure at only the fork or join transitions.

Figure 9 shows the CMMGs obtained from the \(\Rightarrow^e\)'s in Fig. 6.

3.2.7 Conversion into State Machines

Algorithm 3 shows an algorithm for converting a CMMG into a state machine. In the algorithm, \(\text{fired}\), is a Boolean variable and it is added as an attribute of the corresponding class. If \(A(t)\) is a callee’s event, then \(\text{Event}(t)\) is the event; otherwise, \(\text{Event}(t) = \tau\). \(\text{Constraint}(t)\) returns the guard conditions as follows:

\[ \text{Constraint}(t) = \begin{cases} 
\wedge_{\text{REG}(t)} \text{fired}, & \text{if } G(t) \neq \emptyset, \text{ and} \\
\varepsilon & \text{otherwise} \end{cases} \]

where \(\varepsilon\) stands for empty expression. \(\text{Behavior}(t)\) yields the
Algorithm 3: Converting CMMG to a state machine

Input: CMMG \((P, T, F, G, A)\)
Output: State machine \((V, R, r', \Theta, E, C, B)\), Attribute \(\mathcal{A}\)

begin
1. \(T \leftarrow T \setminus \{t \mid \text{Empty}(t), \tau \in I_t \mid = 1\};\)
2. \(\mathcal{A} \leftarrow \{\text{fired}, t \mid t \in T\};\)
3. \(E \leftarrow \{\text{Event}(t) \mid t \in T\};\)
4. \(C \leftarrow \{\text{Constraint}(t) \mid t \in T\};\)
5. \(B \leftarrow \{\text{Behavior}(t) \mid t \in T\};\)
6. \(V \leftarrow \emptyset, R \leftarrow \emptyset, r' \leftarrow \text{new Region};\)
7. \(\text{RNG}(p_s, r, p_e);\)
8. \(\text{I}p \leftarrow \text{new InitialPseudoState}(); \Theta(ip) \leftarrow r;\)
9. if \(\text{Event}(t) = \text{Constraint}(t) = e\) then
   \(s \leftarrow ip;\)
   \(\text{new Transition}(ip, r, \varepsilon, e, s);\)
10. \(t \leftarrow p_s;\)
11. \(\text{while Empty}(t) \wedge \| t \| = 1 \text{ do}\)
12. \(I_p = t \cdot \varepsilon;\)
13. \(\text{while Empty}(t) \wedge \| t \| = 1 \text{ do}\)
14. \(t = t \cdot \varepsilon;\)
15. \(ip \leftarrow \text{new InitialPseudoState}(); \Theta(ip) \leftarrow r;\)
16. \(\text{if Event}(t) = \text{Constraint}(t) = e\) then
   \(s \leftarrow ip;\)
   \(\text{new Transition}(ip, r, \varepsilon, e, s);\)
17. \(t \# \text{null do}\)
18. \(ev \leftarrow \text{Event}(t); \text{const} \leftarrow \text{Constraint}(t);\)
19. \(\text{beh} \leftarrow \text{Behavior}(t);\)
20. if \(t = I_p\) then
21. \(s' \leftarrow \text{new FinalState}();\)
22. \(\text{new Transition}(s, ev, const, beh, s');\)
23. \(\text{return} ;\)
24. if \(\| t \| \geq 2\) then
25. \(s' \leftarrow \text{new CompositeState}(); \Theta(s') \leftarrow r;\)
26. \(\text{for all the } p' \in t \text{ do}\)
27. \(r' \leftarrow \text{new Region}(); \Theta(r') \leftarrow s;\)
28. \(p'^* \leftarrow \text{FJ}(t) \cap \text{Succ}(p');\)
29. \(\text{RNG}(p', r', p'^*);\)
30. \(t = \text{FJ}(t);\)
31. \(s' \leftarrow \text{new SimpleState}(); \Theta(s') \leftarrow r;\)
32. \(t = t \cdot \varepsilon;\)
33. \(\text{new Transition}(s, ev, const, beh, s');\)
34. \(s \leftarrow s';\)
end.

following effect:

\[
\begin{align*}
\text{Behavior}(t) = \begin{cases} 
c.m() & \text{if } A(t) = \text{Sm}, 
send m to c & \text{if } A(t) = \text{!m, } m \in M_{\text{call}}. 
\text{reply to m} & \text{if } A(t) = \text{!m, } m \in M_{\text{reply}}, 
\text{fired}_i = \text{true} & \text{if fired}_i \in \mathcal{A}, \text{ and } 
\varepsilon & \text{otherwise,}
\end{cases}
\end{align*}
\]

where the first three expressions show a synchronous call, an asynchronous call, and a reply to a synchronous call in cbUML and \(c\) is the callee service. Note that the first three conditions and the fourth condition are not alternatives. The predicate \(\text{Empty}(t)\) is true if and only if \(\text{Event}(t) = \tau, \text{Constraint}(t) = e,\) and \(\text{Behavior}(t) = e.\) The "new" expres-

pression shows that a new vertex or region is generated, and it is added to \(V\) or \(R.\)

Procedure \(\text{RNG}(p_s, r, p_e)\) constructs a state machine in region \(R\) for a subnet between places \(p_s\) and \(p_e.\) Lines from 11 to 14 eliminate the transitions that do not have any function. Because a transition from an initial pseudo-state cannot have any triggers and guards [9], a simple state is inserted (lines 19 and 20). Lines 21 to 41 generate states and transitions. When the end of the region is reached, a final state is generated (lines 23 to 26). Lines 27 and 28 also eliminate the transitions that do not have any function. If a transition has more than one output place, procedure \(\text{RNG}(\cdot)\) is called recursively (lines 29 to 35). \(\text{Succ}(p')\) in line 33 returns the set of successors of place \(p'.\) As shown in Fig. 4, calling a synchronous operation and receiving its reply message is represented by a single transition in cbUML; therefore, the succeeding transition that receives the reply message is skipped at line 37.

In Algorithm 3, every transition is visited exactly once because no bridge exists in a CMMG. Algorithm 3 terminates for any CMMG because the set of transitions is finite.

Lemma 5: A CMMG is directly convertible to a state machine.

Algorithm 3 does not change the behavior of services; thus, we can obtain the following theorem.

Theorem 1: Let \(SM\) be the set of state machines synthe-
sized by the CSCB method from choreography $CD$. Then, $SM$ is a weak realization of $CD$: $C(CD) \supseteq C(SM)$.

The state machines shown in Fig. 10 are synthesized by performing Algorithm 3.

4. Computational Evaluation

We developed a prototypical tool of the CSCB method and the projection method in [8] and evaluated the CSCB method in several examples.

The projection method synthesizes flat state machines from the precedence relation. Figure 11 shows an example, which is the flat state machine for Service4. Assume that the number of events related to service $c$ is $|\Sigma_c|$, then, the number of states of the state machine becomes $2^{|\Sigma_c|}$. This method, however, generates numerous unreachable states from the initial state. In this paper, state machines are used after these unreachable states have been removed and the reply message to a synchronous call is considered an asynchronous message that is independent of the synchronous call. Therefore, transitions that receive reply message Check5_rep appears.
Table 1 shows the results and Figs. 1, 2, 12–17, where Figs. 1 and 2 correspond to ex6, show the communication diagrams and dependency relations used in the evaluation. The CSCB method is better than the projection method in all seven examples in terms of the several metrics for intelligibility put forth by Cruz-Lemus et al.

5. Conclusion

In this paper, we discussed an approach to CRP, considering the intelligibility of state machines. We proposed a method for synthesizing state machines without generating state spaces from the choreography defined by a single communication diagram and showed the weak realizability of the proposed method. We also evaluated the proposed method by using metrics about the intelligibility of state machines.

The proposed method does not assure the maximality of synthesized state machines. Synthesizing maximal state machines is a promising topic for future research.

With regard to the intelligibility of synthesized state machines, we believe that there may be some room for improvement. A method for synthesizing state machines from multiple communication diagrams is also left for future research.

References


Toshiyuki Miyamoto received his B.E. and M.E. degrees in electronic engineering from Osaka University, Japan in 1992 and 1994, respectively. Moreover, he received Dr. of Eng. degree in electrical engineering from Osaka University, Japan in 1997. From 2000 to 2001, he was a visiting researcher in Department of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA. Currently, he is an Associate Professor with the Division of Electrical, Electronic and Information Engineering, Osaka University. His areas of research interests include theory and applications of concurrent systems and multi-agent systems. He is a member of IEEE, SICE, and ISCIIE.

Yasuwo Hasegawa received his B.E. degree in electronic and information engineering from Osaka University, Japan in 2010 and his M.E. degree in electrical, electronic, and information engineering from Osaka University, Japan in 2012.

Hiroyuki Oimura received his B.E. degree in electronic and information engineering from Osaka University, Japan in 2013.