A Petri Net Approach to Synthesize Intelligible State Machine Models from Choreography*

Toshiyuki Miyamoto[1] and Yasuwo Hasegawa[1]

Graduate School of Engineering, Osaka University,
Suita, Osaka 565-0871, Japan
miyamoto@eei.eng.osaka-u.ac.jp

Abstract. Application of service-oriented architecture, which builds the entire system by a combination of independent software components, to a wide variety of computer systems is expected. The problem to synthesize state machine models of the services from a communication diagram representing the overall specifications of service interaction is known as the choreography realization problem. It should be minded on automatic synthesis that software models should be simple to be understood easily by software engineers. In this paper, we propose a method to synthesize hierarchical state machine models for the choreography realization problem. The proposed method is evaluated using a metric for intelligibility.

1 Introduction

In recent years, the internationalization of activities and information technology in the enterprise has intensified competition between companies. Companies are under pressure to respond quickly to business needs, and the period for making changes to existing business and launching new businesses has been shortened. For this reason, the need to change or build quickly information systems has been increasing.

Under such circumstances, service-oriented architecture (SOA)[12] has been attracting attention as the architecture of information systems in the enterprise. 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. It is not easy for the designers to design a concrete model directly from requirements since there exists huge gaps. But, defining an abstract specification is relatively simple. Therefore, if we can automatically synthesize a concrete model from abstract high-quality specification, it is expected that designer's workload is greatly reduced and product quality is improved.

In the field of software engineering, there exist several investigations that synthesize the concrete model from the abstract specification. Harel et al. proposes a methodology for synthesizing statechart models from scenario-based requirements[5]. Whittle et al. proposes a methodology for synthesizing hierarchical

* This work was supported by KAKENHI (23500045).

state machine models from expressive scenario descriptions[13]. Liang et al. defines a set of comparison criteria, and surveys 21 different synthesis approaches presented in literature based on the criteria[7]. In addition, the theory of regions has been attracting attention as a method to synthesize nets[3].

In SOA, the problem to synthesize the concrete model from an abstract specification is known as the choreography realization problem[11]. In which the abstract specification, called *choreography*, is defined as a set of interactions among services, which are given in a dependency relation of messages sent and received; the concrete model is called the *service implementation* which defines the behavior of the service. This paper utilizes the communication diagram and the state machine of UML 2.x[10] to describe the choreography and the service implementation, respectively.

Bultan and Fu formally introduced the choreography realization problem in [2]. They used collaboration diagrams of UML1.x and showed some conditions for a given choreography to be realizable. In addition, they showed a method to represent the service implementation as the state space in which a state was defined as a set of unsent messages, and they also showed a method to map to a set of finite state machines. However, it is not intelligible because the number of states increases exponentially as the number of messages increases. Furthermore, they have adopted the semantics that message send and receive events for a synchronous call occur simultaneously. Under this 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"[10] can not be represented.

Miyamoto et al. have proposed a method to synthesize hierarchical state machines from the choreography given in communication diagrams[8]. In the method, dependency constraints between message send and receive events are represented by Petri nets[9]; the state machine is synthesized from its reachability space. A method to extract the hierarchical structure by analyzing the reachability space is given, but the technique can only be applied to simple cases.

This paper proposes a method of converting a Petri net into the state machine directly. It is shown that there is a relation between the possibility of direct conversion and structural properties of Petri nets. At first the proposed method converts the Petri net so as to satisfy the structural properties, then it converts Petri nets into hierarchical state machines without generating their state spaces.

This paper is organized as follows. Section 2 introduces an UML subset, called *subset of UML for formally describing choreography and behavioral feature* (cbUML), to discuss the choreography realization problem, and an extended Petri net, called message mark graph (MMG). The proposed method, called Construct State-machine Cutting Bridges (CSCB) method, is evaluated in terms of the intelligibility in Sect. 3. However, in this paper it is assumed that the choreography is given in a single communication diagram as the first stage of the study. Section 4 is the conclusion.

2 Preliminaries

2.1 cbUML

Let us introduce a subset of UML, called cbUML, for the discussion in this paper.

Definition 1 (cbUML). *cbUML is a tuple* $(\mathcal{C}, \mathcal{M}, \mathcal{A}, \mathcal{CD}, \mathcal{SM})$*, where* $\mathcal{C}$ *is the set of classes,* $\mathcal{M}$ *is the set of messages,* $\mathcal{A}$ *is the set of attributes,* $\mathcal{CD}$ *is the set of communication diagrams, and* $\mathcal{SM}$ *is the set of state machines. Each of messages and attributes is owned by a class, and behavior of a class is defined by a state machine.*

Messages The set $\mathcal{M}$ of messages are partitioned with respect to the sort of messages: $\mathcal{M} = \mathcal{M}_{sop} \cup \mathcal{M}_{aop} \cup \mathcal{M}_{rep}$, where $\mathcal{M}_{sop}$ is the set of *synchronous messages* generated by synchronous calls, $\mathcal{M}_{aop}$ is the set of *asynchronous messages* generated by asynchronous calls, and $\mathcal{M}_{rep}$ is the set of *reply messages* to synchronous messages. Let $\mathcal{M}_s = \mathcal{M}_{sop}$ and $\mathcal{M}_a = \mathcal{M}_{aop} \cup \mathcal{M}_{rep}$. Correspondence between the synchronous call and its response is given by the function $refer : \mathcal{M} \mapsto \mathcal{M} \cup \{\text{nil}\}$, such that $\forall m \in \mathcal{M}_{sop} : refer(m) \in \mathcal{M}_{rep}$, $\forall m \in \mathcal{M}_{rep} : refer(m) \in \mathcal{M}_{sop}$, and $\forall m \in \mathcal{M}_{aop} : refer(m) = \text{nil}$. Note that $\forall m \in \mathcal{M}_{sop} \cup \mathcal{M}_{rep} : refer(refer(m)) = m$.

There is a difference in behavior during interactions due to differences in the sort of message as follows: In a synchronous call, caller's execution is stopped until it receives a reply from the callee. On the other hand, in the asynchronous call, the caller is possible to continue to operate, regardless of the behavior of the callee side.

In UML, each message has two events: a *send event* and a *receive event*. For a synchronous message, it is considered that they occur simultaneously. However, for the later discussion we need two events that occur separately. 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. A preparation event and a send-receive event for a synchronous message $m \in \mathcal{M}_s$ are denoted by $\$m$ and $!m$, respectively. For an asynchronous message $m \in \mathcal{M}_a$, its send event and receive event are denoted by $!m$ and $?m$, respectively. Hereafter a send event is a send-receive event for a synchronous message or a send event for an asynchronous message. The set Σ of message events and the set Δ of send events are defined as follows:

$$\begin{aligned}
&\Sigma = \{\$m, !m \mid m \in \mathcal{M}_s\} \cup \{!m, ?m \mid m \in \mathcal{M}_a\}, \text{and} \\
&\Delta = \{!m \mid m \in \mathcal{M}\}.
\end{aligned}$$

Communication Diagrams Communication diagrams show interactions where the arcs between the communicating lifelines are decorated with description of the passed messages and their sequencing.

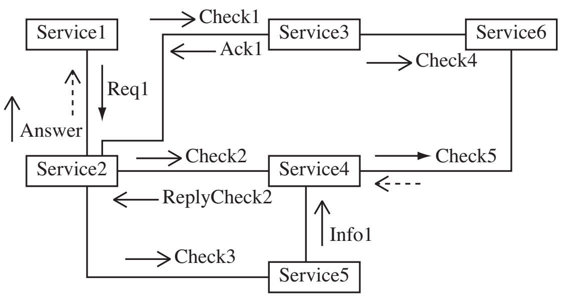

Fig. 1. A communication diagram

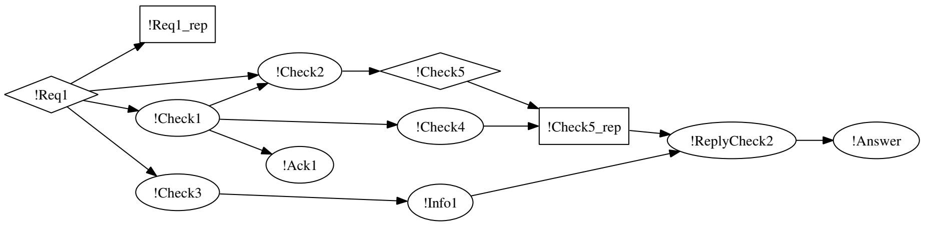

Fig. 2. D_{cd}

Definition 2 (Communication Diagram). *A communication diagram* $cd \in \mathcal{CD}$ *is a tuple* $cd = (\mathcal{C}_{cd}, \mathcal{M}_{cd}, Conn_{cd}, line_{cd}, D_{cd})$*, where* $\mathcal{C}_{cd} \subseteq \mathcal{C}$ *is the set of instances of classes (called lifelines or objects) in cd,* $\mathcal{M}_{cd} \subseteq \mathcal{M}$ *is the set of messages in cd,* $Conn_{cd} \subseteq \mathcal{C}_{cd} \times \mathcal{C}_{cd}$ *is the set of connectors, which is given as a symmetric relation on* $\mathcal{C}_{cd}$*,* $line_{cd} : \mathcal{M}_{cd} \mapsto Conn_{cd}$ *assigns a connector for each message, and* $D_{cd} \subseteq \Delta_{cd} \times \Delta_{cd}$ *is a dependency relation among send events. Note that the reflexive and transitive closure of* D_{cd} *is a partial order.*

A *conversation* is the sequence of messages exchanged among the objects. The set of conversations defined by a communication diagram *cd* is denoted by $\mathfrak{C}(cd) \subseteq 2^{\mathcal{M}^*}$, where $\mathcal{M}^*$ is the set of all sequences of messages.

Definition 3. *A conversation* $\sigma = m_1 m_2 \cdots m_n$ *is in* $\mathfrak{C}(cd)$ *if and only if* $\sigma \in \mathcal{M}^*$ *and the corresponding sequence* $\gamma = !m_1 !m_2 \cdots !m_n$ *of send events satisfies* $\forall i, j \in [1..n] : (!m_i, !m_j) \in D_{cd} \Rightarrow i < j$.

Figure 1 shows a communication diagram. In this example, messages Req1 and Check5 are synchronous messages, and dashed lines with open arrow head are their reply messages. Suppose that the dependency relation among send 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 the example.

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

State Machines The behavior of each object is described by a state machine.

Definition 4 (State Machine). *A state machine is a tuple* $sm = (V_{sm}, R_{sm}, top_{sm}, cont_{sm}, TR_{sm}, E_{sm}, Const_{sm}, Beh_{sm})$*, where* $V_{sm} = SS_{sm} \cup CS_{sm} \cup FS_{sm} \cup IS_{sm}$ *is the set of vertices*[1]*,* R_{sm} *is the set of regions,* $top \in R_{sm}$ *is the top region,* $cont_{sm} : (V_{sm} \cup R_{sm}) \setminus \{top_{sm}\} \mapsto (CS_{sm} \cup R_{sm})$ *is an ownership relation between vertices and regions,* TR_{sm} *is the set of transition relations,* E_{sm} *is the set of events,* $Const_{sm}$ *is the set of constraints, and* Beh_{sm} *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. A composite state is able to own one or more regions, and a region is able to own vertices. The function $cont_{sm}$ represents the ownership of vertices and regions, and $cont_{sm}(x_1) = x_2$ means that x_1 is owned by x_2. For a $x \in V_{sm} \cup R_{sm}$, let $des(x) = \{x' \mid \exists i > 0 : cont^i_{sm}(x') = x\}$ be the set of descendants of x, where $cont^1_{sm}(\cdot) = cont_{sm}(\cdot)$ and $cont^i_{sm}(\cdot) = cont_{sm}(cont^{i-1}_{sm}(\cdot))$ $(i > 1)$.

Definition 5 (Orthogonal State). *If there exist vertices* $v_1, v_2 \in V_{sm}$ *and different regions* $r_1, r_2 \in R_{sm}$*,* $r_1 \neq r_2$ *such that* $cont_{sm}(r_1) = cont_{sm}(r_2)$*,* $v_1 \in des(r_1)$*, and* $v_2 \in des(r_2)$*, two vertices* v_1, v_2 *are called orthogonal, and denoted by* $v_1 \perp v_2$*.*

Definition 6. *A set* $\hat{V}_{sm} \subset V_{sm}$ *of vertices is called consistent if and only if for each pair* $v_1, v_2 \in \hat{V}_{sm}$ *of vertices* $v_1 \perp v_2$*,* $v_1 \in des(v_2)$*, or* $v_2 \in des(v_1)$*.*

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

A transition relation $etr \in TR_{sm}$ is a tuple $etr = (src, trig, grd, eff, tgt)$, where $src \in V_{sm}$ is the originating vertex of the transition, a trigger $trig \in E_{sm}$ is the event that makes the transition fire, a guard $grd \in Const_{sm}$ is a constraint, an effect $eff \in Beh_{sm}$ is an optional behavior to be performed when the transition fires, and $tgt \in V_{sm}$ is the target vertex. Note that $\{src, tgt\}$ must not be consistent. According to the UML specification[10], triggers, guards, and effects are denoted like "$trig[grd]/eff$" in diagrams. It is supposed that $\Sigma_{sm} \subseteq Beh_{sm}$, and a caller's event of message sending becomes an effect and a callee's event becomes a trigger.

Due to space limitations, the details of operational semantics of state machines are omitted, and the steps of doing synchronous calls and asynchronous calls are explained by examples.

Figure 3 shows the execution of the asynchronous call. Gray states are active. When state machine sm1 transitions from state s11 to state s12 by the occurrence of the completion event, an asynchronous call is executed. At this time the send

[1] SS_{sm} is the set of *simple states*, CS_{sm} is the set of *composite states*, FS_{sm} is the set of *final states*, and IS_{sm} is the set of *initial pseudo states.*

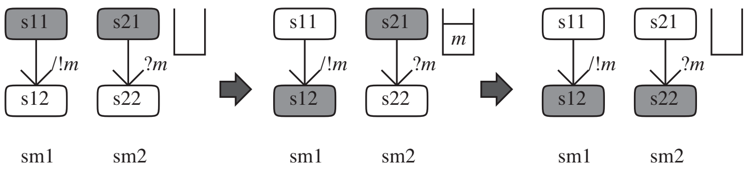

Fig. 3. Steps for an asynchronous call

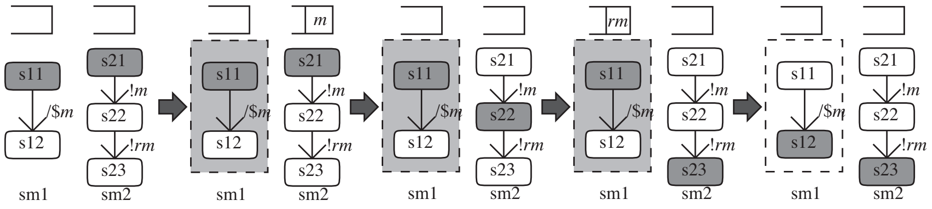

Fig. 4. Steps for a synchronous call

event $!m$ occurs, and the message m will be appended at the end of the queue of sm2. The state machine sm2 transitions from state s21 to state s22 consuming the message m by the occurrence of the receive event $?m$.

Figure 4 shows the execution of the 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. In addition, the message m is appended at the end of the queue of sm2. The state machine sm2 transitions from state s21 to s22 consuming the message m by the occurrence of the send-receive event $!m$. The sm2 sends a reply message rm to sm1 on transitioning from s22 to s23. At this time the send event $!rm$ occurs, and the message rm is appended at the end of the queue of sm1. The sm1 releases the suspended region, and transitions from state s11 to state s12 consuming the 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 $\mathcal{SM}$ of state machines is denoted by $\mathfrak{C}(\mathcal{SM})$.

2.2 Petri nets

The proposed method represents the dependency relation between message send an receive events by using Petri nets[9]. Since this paper assumes that the choreography is given by a communication diagram, Petri nets that appear in this paper are marked graphs.

A Petri net $N = (P, T, F, W)$ is called *ordinary* when $\forall(x, y) \in F : W(x, y) = 1$. Then the weight function W is omitted. 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 $\bullet x$. In the same way, the set $\{y \mid (x, y) \in F\}$ is called the postset of x, and denoted by $x\bullet$.

A place $p \in P$ is called a *source* place and a *sink* place when $\bullet p = \emptyset$ and $p\bullet = \emptyset$, respectively. In the same way 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 $|t\bullet| \geq 1$ and $|\bullet t| \geq 1$, respectively. The sets of join transitions and fork transitions are denoted by T_{join} and T_{fork}, respectively. Under the standard definition, 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 $\forall p \in P : |\bullet p| \leq 1$ and $|p\bullet| \leq 1$.

Definition 7 (Message Marked Graph). *A message marked graph (MMG) is a tuple $N = (P, T, F, W, G, A)$, where the underlying Petri net (P, T, F, W) satisfies the following conditions:*

1. *N is an ordinary and acyclic,*
2. *there exist only one source place p_s and only one sink place p_e,*
3. *no source transitions and sink transitions exist, and*
4. *$|p_s\bullet| = 1$, $|\bullet p_e| = 1$, and $\forall p \in P\backslash\{p_s, p_e\} : [|\bullet p| = 1, |p\bullet| = 1]$.*

$G : T \mapsto 2^T$ is a firing constraint, and the partial function $A : T \mapsto \Sigma$ assigns an event for the transition.

A state of MMG is expressed by a pair (M, X), where $M : P \mapsto \mathbb{Z}^+$ is a marking and $X : T \mapsto \mathbb{B}$ is a firing configuration, where $\mathbb{Z}^+$ is the set of non-negative integers and $\mathbb{B} = \{\text{true}, \text{false}\}$. The initial state (M_0, X_0) of MMG is given as follows:

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

A transition $t \in T$ is enabled if and only if $\forall p \in \bullet t : M(p) \geq W(p, t)$ and $\forall t' \in G(t) : X(t') = \text{true}$. A new state (M', X') obtained by the firing of transition t is given as follows:

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

Handles and Bridges Let $N = (P, T, F)$, and $N_1 = (P_1, T_1, F_1)$ be a subnet of N. An elementary path $H = (n_1, \ldots, n_r), r \geq 2$ of N is a *handle* of N_1 if and only if $H \cap (P_1 \cup T_1) = \{n_1, n_r\}$.

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 $B = (n_1, \ldots, n_r), r \geq 2$ is a *bridge* from N_1 to N_2 if and only if $B \cap (P_1 \cup T_1) = \{n_1\}$ and $B \cap (P_2 \cup T_2) = \{n_r\}$.

For a transition $t \in T$, $FJ(t) \subseteq T$ is a set of terminal vertices of handles starting from t. Similarly, $JF(t) \subseteq T$ is a set of starting vertices of handles terminating at t. Please refer to [4] for more detail about handles and bridges.

Convertible MMG Intuitively, if two states in a state machine are consistent, both states may be active concurrently. The UML specification prohibits drawing a transition between consistent states. An MMG is a representation of the relationship between the order of the messages in the marked graph; in general cases a state machine which satisfies the specification can not be directly, namely without generating its state space, derived from the MMG.

Let us introduce a subclass of MMG called the convertible MMG (CMMG), from which we can get a state machine satisfying the specification directly by using Algorithm 1 shown later.

Definition 8. *A MMG is called a CMMG if the following conditions hold:*

1. $|T_{fork}| = |T_{join}|$
2. $T_{fork} \cap T_{join} = \emptyset$
3. *For any pair of handles H, H' in the MMG, only one of following conditions holds: (a) H and H' share the same starting vertex and the same terminal vertex, and (b) the starting vertices of H and H' are different and the terminal vertices of H and H' are different.*
4. *If $A(t) = \$m$, then $t \bullet\bullet = \{t'\}$, $A(t') = ?refer(m)$.*

From the definition of CMMG, for all $t \in T_{fork}$ (resp. $t \in T_{join}$) $|FJ(t)| = 1$ (resp. $|JF(t)| = 1$). Moreover the following lemma holds.

Lemma 1. *Let N be a MMG, N_1 be a subnet of N, and H be a handle of N_1. If N is a CMMG, then there is no bridge from H to N_1.*

Algorithm 1 shows how a CMMG is converted to a state machine. In the algorithm, if $A(t) \in \{!m \mid m \in M_s\} \cup \{?m \mid m \in M_a\}$ then $Event(t) = A(t)$, and $Constraint(t) = \wedge_{a \in G(t)} fired_a$. The mapping $Behavior(t)$ is given as follows:

$$Behavior(t) = \begin{cases} own(m).m(\cdots) & \text{if } A(t) \in \{\$m \mid m \in \mathcal{M}_{sop}\} \\ \text{send } m(\cdots) \text{ to } own(m) & \text{if } A(t) \in \{!m \mid m \in \mathcal{M}_{aop}\} \\ \text{reply to } m(\cdots) & \text{if } A(t) \in \{!m \mid m \in \mathcal{M}_{rep}\} \end{cases}$$

where, $own(m)$ is the owner object of message m, and in cbUML these expressions show a synchronous call, an asynchronous call, and a reply for a synchronous call. In addition, if $fired_t \in A$, then add an expression '$fired_t =$ true' to $Behavior(t)$. The 'new' expression shows a new element is generated.

Lemma 2. *A CMMG is directly convertible to a state machine.*

Figure 5 shows an example of CMMG, and Fig. 6 is the synthesized state machine by Algorithm 1. In Fig. 5, places on each edge, p_s and p_e are omitted.

3 Choreography Realization Problem

3.1 Choreography Realization Problem

By a single communication diagram, one scenario that is an interaction of objects in the system are described. All behavior of the system is given by a set of communication diagram; this is referred to as choreography.

Algorithm 1: Converting CMMG to a state machine

```
Input: CMMG $(P, T, F, G, A)$
Output: State machine $(V, R, top, cont, TR, E, Const, Beh)$, Attribute $\mathcal{A}$
begin
    $\mathcal{A} \leftarrow \{fired_t \mid t \in \bigcup_{t' \in T} G(t')\}$;
    $E \leftarrow \{Event(t) \mid t \in T\}$;
    $Const \leftarrow \{Constraint(t) \mid t \in T\}$;
    $Beh \leftarrow \{Behavior(t) \mid t \in T\}$;
    $V \leftarrow \emptyset$;
    $R \leftarrow \emptyset$;
    $t_{\text{init}} \leftarrow p_s \bullet$;
    $t_{\text{end}} \leftarrow \bullet p_e$;
    $top \leftarrow$ new Region();
    RNG($t_{\text{init}}, top, t_{\text{end}}$);
RNG($t, r, t_e$)
    $ip \leftarrow$ new InitialPseudoState(); $cont(ip) \leftarrow r$;
    if $Event(t) = Constraint(t) = \varepsilon$ then
        $s \leftarrow ip$
    else
        $s \leftarrow$ new SimpleState(); $cont(s) \leftarrow r$;
        new Transition $(ip, \varepsilon, \varepsilon, \varepsilon, s)$;
    while $t \neq t_e$ do
        $ev \leftarrow Event(t)$; $const \leftarrow Constraint(t)$; $beh \leftarrow Behavior(t)$;
        if $ev = const = beh = \varepsilon \wedge |t \bullet| = 1$ then
            $t \leftarrow t \bullet \bullet$; continue;
        if $A(t) \in \{\$m \mid m \in \mathcal{M}_s\}$ then $t \leftarrow t \bullet \bullet$;
        if $|t \bullet| \geq 2$ then
            $s' \leftarrow$ new CompositeState(); $cont(s') \leftarrow r$;
            forall the $p' \in t\bullet$ do
                $r' \leftarrow$ new Region(); $cont(r') \leftarrow s$;
                $RNG(p'\bullet, r', FJ(t))$;
            $t \leftarrow FJ(t)$ ;
        else
            $s' \leftarrow$ new SimpleState(); $cont(s') \leftarrow r$;
            $t \leftarrow t \bullet \bullet$;
        new Transition $(s, ev, const, beh, s')$;
        $s \leftarrow s'$;
    $fs \leftarrow$ new FinalState();
    new Transition $(s, \varepsilon, \varepsilon, \varepsilon, fs)$;
```

Intuitively, the choreography realization problem is the problem to determine whether it is possible to synthesize a set of state machines which realize the choreography. In addition, it is desired to synthesize the state machines. The choreography realization problem is formally defined as follows.

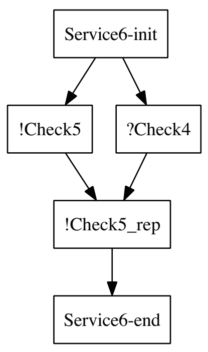

Fig. 5. CMMG

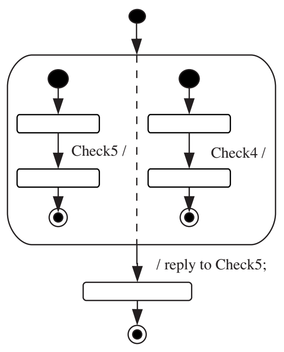

Fig. 6. Generated state machine

Problem 1. For a given set $\mathcal{CD}$ of communication diagrams, is it possible to synthesize the set $\mathcal{SM}$ of state machines which satisfy $\mathfrak{C}(\mathcal{CD}) = \mathfrak{C}(\mathcal{SM})$? If possible, obtain the set of state machines.

In the case of un-realizable choreography, is is desired to synthesize state machines which behave as close to the choreography as possible. It is called weakly realizable if there exist state machines which satisfy $\mathfrak{C}(\mathcal{CD}) \supseteq \mathfrak{C}(\mathcal{SM})$. For a weakly realizable choreography, obtain the set of state machines whose $\mathfrak{C}(\mathcal{SM})$ is maximal.

In [2], sufficient realizability conditions for a class of collaboration diagrams have been shown. We suppose that given $\mathcal{CD}$ is (weak) realizable hereafter and the set $\mathcal{CD}$ contains only one communication diagram.

3.2 CSCB Method

The proposed CSCB method synthesizes state machines from a communication diagram as below. Due to space limitations the details of the algorithm are omitted.

1. Construct a dependency relation $\Rightarrow_{cd}$ on the set of events.
   For each object c, perform the following steps.
2. Derive a dependency relation $\Rightarrow_{cd}^{c}$ from $\Rightarrow_{cd}$.
3. Construct an MMG from $\Rightarrow_{cd}^{c}$.
4. Cut T-T bridges from the MMG.
5. Separate fork and join transitions in the MMG.
6. Find one-to-one correspondence between T_{fork} and T_{join} in the MMG.
7. Perform Algorithm 1.

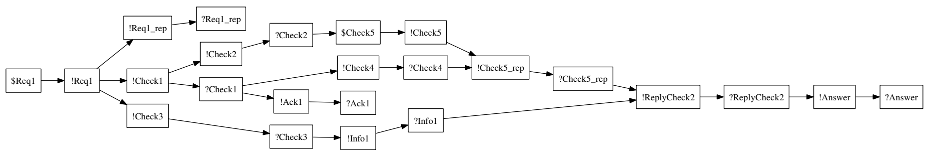

Fig. 7. $\Rightarrow_{cd}$

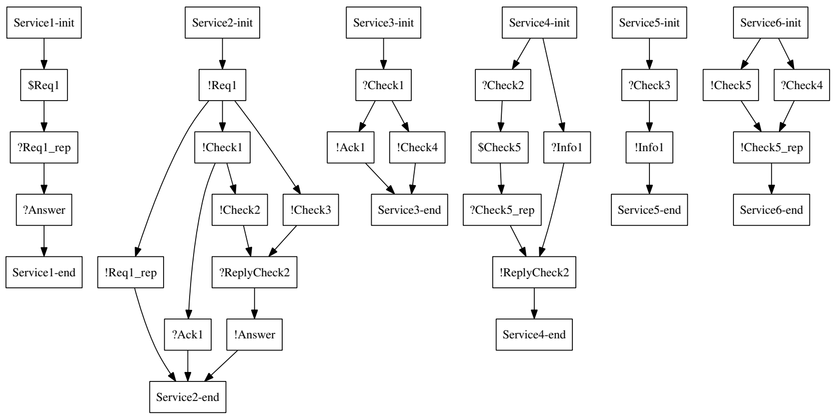

Fig. 8. $\Rightarrow^{c}_{cd}$ and MMGs for Service1, Service2, ... are shown from left to right.

Construction of dependency relation $\Rightarrow_{cd}$ The dependency relation $\Rightarrow_{cd} \subseteq \Sigma_{cd} \times \Sigma_{cd}$ on the set of events is given by the following expression:

$$\begin{aligned}\Rightarrow_{cd} = D_{cd} &\cup \{(\$m, !m) \mid m \in \mathcal{M}_s^{cd}\} \cup \{(!m, ?m) \mid m \in \mathcal{M}_a^{cd}\}\\ &\cup \{(?m_1, !m_2) \mid m_1 \in \mathcal{M}_a^{cd}, m_2 \in \mathcal{M}_a^{cd}, (!m_1, !m_2) \in D_{cd}\}\\ &\cup \{(?m_1, \$m_2) \mid m_1 \in \mathcal{M}_a^{cd}, m_2 \in \mathcal{M}_s^{cd}, (!m_1, !m_2) \in D_{cd}\}\\ &\cup \{(!m_1, \$m_2) \mid m_1 \in \mathcal{M}_s^{cd}, m_2 \in \mathcal{M}_s^{cd}, (!m_1, !m_2) \in D_{cd}\}\end{aligned}$$

Figure 7 shows the dependency relation $\Rightarrow_{cd}$ for the communication diagram shown in Fig. 1.

Deriving $\Rightarrow^{c}_{cd}$ and Construction of MMG At first, $\Rightarrow_{cd}$ is transitively reduced, then the dependency relation $\Rightarrow^{c}_{cd}$ for each object c is derived. At this time, in order to satisfy the condition 4 of CMMG, for all synchronous message $m \in \mathcal{M}_a$, if there exists an event $e \neq ?refer(m)$ such that $(\$m, e) \in \Rightarrow^{c}_{cd}$, then a relation $(?refer(m), e)$ is added in $\Rightarrow^{c}_{cd}$.

The dependency relations $\Rightarrow^{c}_{cd}$, which are derived from the dependency relation $\Rightarrow_{cd}$ shown in Fig. 7, are shown in Fig. 8. Here, since $(\$Req1, ?Req1_rep)$,

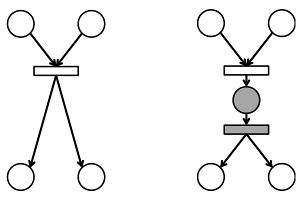
Fig. 9. Separating fork and join transition

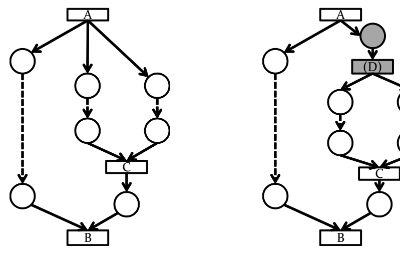

Fig. 10. Finding one-to-one correspondence

$(\$Req1, ?Answer) \in \Rightarrow^{c}_{cd}$ for Service1, a relation $(?Req1_{r}ep, ?Answer)$ is added in $\Rightarrow^{\text{Service1}}_{cd}$.

MMGs are constructed by converting vertices into transitions, adding a place for each edge, and adding source and sink places in Fig. 8.

Cutting T-T bridges As shown in Lemma 1, since bridges are unnecessary in CMMGs, they are cut. In the example in Fig. 8, (!Check1 !Check2 ?ReplyCheck2) of Service2 is a bridge. After removing edges (!Check1, !Check2) and (!Check2, ?ReplyCheck2), edge (Service2-init, !Check2) and (!Check2, Service2-end) are added. At that time, in order to avoid changing the behavior, the following firing conditions are added.

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

Cutting all bridges is not always necessary. Let U be a set of bridges and $f : U \mapsto 2^U$ be a function such that $f(u)$ is a set of bridges which will not be bridges by cutting bridge u. Then, the problem to finding the set of bridges results in the set cover problem[6].

Separating fork and join transitions If there exists fork and join transition, it is split into a fork transition and a join transition as shown in Fig. 9.

Finding one-to-one correspondence As shown in Fig. 10, dummy transition D is added in order to find one-to-one correspondence between fork and join transitions..

Lemma 3. *The MMG obtained by applying steps 1∼6 of CSCB method is a CMMG.*

Figure 11 shows CMMGs obtained from MMG in Fig 8.

Conversion into state machines By performing Algorithm 1, state machines shown in Fig.s 12, 14, 13, 15, 16, and 6 are obtained.

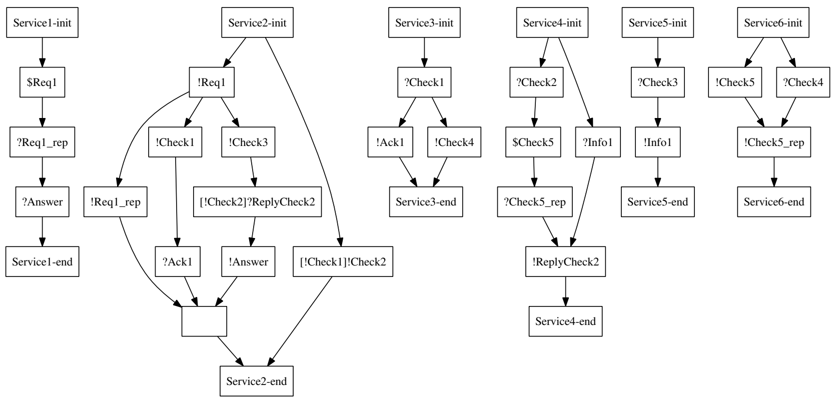

Fig. 11. CMMG of the example

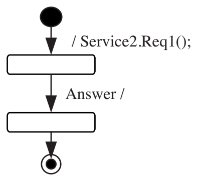

Fig. 12. State machine of Service1

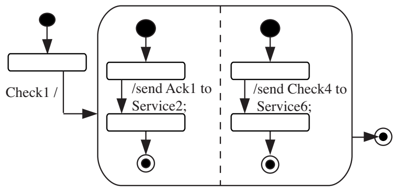

Fig. 13. State machine of Service3

3.3 Intelligibility Evaluation

Antonio et al. have experimentally evaluated the relationship between metrics and intelligibility of the state machines by measuring time to understand state machines[1]. According to the result, state machines are intelligible the smaller the following metrics: the number of simple states (NSS), the number of transitions (NT), and the number of guards (NG). In this section, the CSCB method

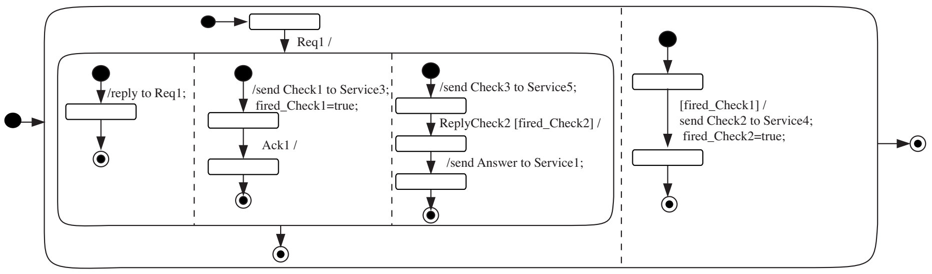
Req1 /
/reply to Req1;
/send Check1 to Service3;
fired_Check1=true;
Ack1 /
/send Check3 to Service5;
ReplyCheck2 [fired_Check2] /
/send Answer to Service1;
[fired_Check1] /
send Check2 to Service4;
fired_Check2=true;

Fig. 14. State machine of Service2

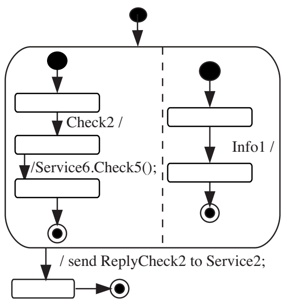

Fig. 15. State machine of Service4

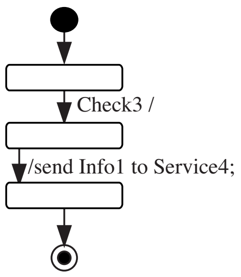

Fig. 16. State machine of Service5

Table 1. Evaluation result

	Method in [2]			Method in [8]			CSCB		
	NSS	NT	NG	NSS	NT	NG	NSS	NT	NG
Service1	5	7	0	5	9	0	2	3	0
Service2	31	59	0	31	59	0	9	17	2
Service3	5	7	0	5	9	0	5	9	0
Service4	8	11	0	7	11	0	6	10	0
Service5	3	4	0	3	4	0	3	4	0
Service6	5	7	0	5	9	0	5	9	0

is evaluated by comparing with Bultan's method[2], the state space generation method[8] by using the above metric.

The Bultan's method[2] synthesize flat state machines from the dependency relation. Suppose the number of events relating to object c to be $|\Sigma^c|$, then the number of states of the state machine becomes $2^{|\Sigma^c|}$. This method, however, generates plenty of unreachable state from the initial state. In this paper, state machines after removing these unreachable states are used.

The state space generation method[8] generates a state space for each MMG at first, and then converts the state spaces into state machines. The method, however, tries to find "independent sequences" in the state space, and tries to reduce the number of states by using composite states. Therefore, when no independent sequence is found, the same result with the Bultan's method is obtained.

Table 1 shows values of the metrics of state machines which are obtained from the communication diagram in Fig. 1. Note that in the Bultan's method and the state space generation method, the reply message to a synchronous call is considered as an asynchronous message which is independent with the synchronous call. On the other hand, in the proposed method, the state transition relating to a synchronous call terminates only when it receives the reply message. The proposed method adds relation at step2 so as each preparation event for message sending has only the receive event of the reply message as an immediate successor. Therefore, in the dependency relation for Service1, events ?Req1_rep and ?Answer are in concurrent for the Bultan's method and the state space generation method, but they are in sequential for the CSCB method.

As for Service2, since the state space generation method failed to find independent sequences, the state space are converted into a state machine as is. In contrast, the proposed method succeeds to significantly reduce the number of states by cutting bridges.

4 Conclusion

In this paper, we considered the approach to the choreography realization problem considering intelligibility of synthesized state machines. We proposed a method to synthesize state machines without generating state spaces from the choreography defined by single communication diagram. We evaluated the proposed method by using metrics about intelligibility of the generate state machines.

References

1. Antonio Cruz-Lemus, J., Genero, M., Piattini, M.: Metrics for UML Statechart Diagrams. In: Genero, M., Piattini, M., Calero, C. (eds.) Metrics for Software Conceptual Models, pp. 237–272. Imperial College Press, London (2005)
2. Bultan, T., Fu, X.: Specification of realizable service conversations using collaboration diagrams. Service Oriented Computing and Applications 2(1), 27–39 (2008)
3. Desel, J., Yakovlev, A. (eds.): Proceedings of 2nd Workshop on Application of Region Theory (Jun 2011)
4. Esparza, J., Silva, M.: Circuits, Handles, Bridges and Nets. Lecture Notes in Computer Science 483, 209–242 (1991)
5. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: generating statechart models from scenario-based requirements. In: Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling, pp. 309–324. Springer (Jan 2005)
6. Jungnickel, D.: Graphs, Networks and Algorithms. Springer, 3rd edn. (2007)
7. Liang, H., Dingel, J., Diskin, Z.: A Comparative Survey of Scenario-based to State-based Model Synthesis Approaches. In: 2006 International workshop on Scenarios and state machines: models, algorithms, and tools. pp. 5–11 (2006)
8. Miyamoto, T., Kurahata, H., Fujii, T., Hosokawa, R.: Synthesis of state machine diagrams from communication diagrams using petri nets. Innovations in Systems and Software Engineering 6, 39–46 (2010)
9. Murata, T.: Petri nets: Properties, analysis and applications. Proc. IEEE 77(4), 541–580 (Apr 1989)
10. OMG: Unified modeling language, http://www.uml.org/
11. Su, J., Bultan, T., Fu, X., Zhao, X.: Towards a theory of web service choreographies. In: Proceedings of the 4th international conference on Web services and formal methods. pp. 1–16 (2008)
12. Thomas, E.: Service-Oriented Architecture. Prentice Hall (2004)
13. Whittle, J., Jayaraman, P.K.: Synthesizing hierarchical state machines from expressive scenario descriptions. ACM Transactions on Software Engineering and Methodology 19(3), 1–45 (Jan 2010)