A Method to Convert Concurrent EFSMs with Multi-Rendezvous into Synchronous Sequential Circuit

Akira KITAJIMA¹, Keiichi YASUMOTO¹¹, Nonmembers, Teruo HIGASHINO¹, and Kenichi TANIGUCHI¹, Members

SUMMARY In this paper, we propose a technique to synthesize a hardware circuit from a protocol specification consisting of several concurrent EFSMs with multi-rendezvous specified among their subsets. In our class, each multi-rendezvous can be specified among more than two EFSMs, and several multi-rendezvous can be specified for different combinations of EFSMs. In the proposed technique, using the information such as current states of EFSMs, input values at external gates and guard expressions, we compose a circuit to evaluate whether each multi-rendezvous can be executed. If several exclusive multi-rendezvous get executable simultaneously for some combinations of EFSMs, we select one of them according to the priority order given in advance. We compose such a circuit as a combination logic circuit so that it works fast. By applying our technique to Abracadabra protocol specified in LOTOS, it is confirmed that the derived circuit handles multi-rendezvous efficiently.

key words: communication protocols, multi-rendezvous, concurrent EFSMs, controller synthesis, synchronous sequential circuits

1. Introduction

As the growth of computer networks, the needs for efficient implementation of communication protocols are increasing. For the needs, the techniques for implementing protocols as hardware circuits have been focused in recent years. In general, each protocol has various parameters and several behavior modes which make influence its performance. In the implementation, an effective combination should be selected from a variety of combinations. For such a purpose, rapid prototyping techniques for synthesizing hardware circuits automatically from formal specifications of protocols, are useful in the preliminary phases of developments.

Recently, several researches for synthesizing hardware circuits from protocol specifications in formal specification languages, have been proposed [6], [7], [12], [13]. For example, Ref. [7], [13] have proposed hardware synthesis techniques from SDL and Estelle, respectively. However, they do not deal with the highly structured specifications containing synchronization among concurrent modules like multi-rendezvous (the mechanism to allow multiple concurrent modules to synchronize with each other by exchanging data when some conditions hold among the modules). In Ref. [6], a technique to convert LOTOS[3] specifications to VHDL specifications has been proposed. In the technique, however, each multi-rendezvous is restricted only between two processes. Ref. [1], [12] have also proposed techniques to synthesize hardware circuits from LOTOS specifications. However, they do not treat data values exchanged among processes because they focus only on Basic LOTOS.

Since LOTOS specifications are composed of multiple sub-processes which are dynamically invoked, the combination of synchronizing processes/events may be decided dynamically at each time (it causes inefficiency). For efficient implementation of multi-rendezvous, Ref. [11] has proposed a method to derive all possible multi-rendezvous instances from a LOTOS specification. However, the method requires the complete reachability analysis among all parallel processes, and the number of all possible multi-rendezvous instances is $O(k^n)$ in the worst case where $k$ is the maximal number of transitions for an EFSM and $n$ is the number of EFSMs.

In this paper, we propose a technique to synthesize a hardware circuit from a protocol specification consisting of several concurrent EFSMs with multi-rendezvous specified among their subsets. In our EFSM model, each multi-rendezvous can be specified among more than two EFSMs, and several exclusive multi-rendezvous can be specified for different combinations of EFSMs.

In order to represent all possible multi-rendezvous instances statically, we use the following tuple (rendezvous information): (1) the tuple of EFSMs which synchronize with each other by the rendezvous and (2) the tuple of I/O event (which includes its execution condition or guard expression) sets where every combination of events in the sets can synchronize with each other among the corresponding EFSMs.

In general, the number of rendezvous information is the same as that of rendezvous instances, that is, $O(k^n)$ in the worst case if we do not merge them. In our model, we reduce the number by merging some tuples which have common values exchanged by the rendezvous. This technique makes the number of elements in the multi-rendezvous table only $O(p \cdot n)$ where $p$ is the number of combinations of EFSMs which may perform multi-rendezvous. In our method, for each merged rendezvous information, the part of checking executability of multi-rendezvous for the information is composed. So the size of the part is also $O(p \cdot n)$. In general, $p \cdot n \ll k^n$ holds. This also makes the time for checking each multi-rendezvous more efficiently[5].

In our implementation technique, when concurrent EFSMs with their multi-rendezvous table are given, we compose a circuit to evaluate whether each multi-rendezvous can be

¹The authors are with the Department of Information and Computer Sciences, Osaka University, Toyonaka-shi, 560 Japan.
¹¹The author is with the Faculty of Economics, Shiga University, Hikone-shi, 522 Japan.
executed based on the corresponding rendezvous information (we call the circuit the multi-rendezvous executability check part). If several mutually exclusive multi-rendezvous get executable simultaneously for some combinations of EFSMs, as a simple reasonable solution, we select one of them according to a priority order given in advance. To do so, we compose the circuit called conflict avoidance part. Multi-rendezvous specified among EFSMs can be implemented efficiently by composing the above circuits as a combinational logic circuit.

In Sect. 2, we give a brief outline to convert a LOTOS specification to EFSMs with multi-rendezvous using an example. Section 3 is devoted to explain how the EFSMs with multi-rendezvous behave and what kind of information is required in the multi-rendezvous table to control the EFSMs. We give a technique to compose hardware circuits for the EFSMs and control mechanism among them in Sect. 4. The evaluation results applied to Abracadabra protocol [4] are presented in Sect. 5 and the paper is concluded in Sect. 6.

2. Converting LOTOS Specification to EFSMs with Multi-Rendezvous

2.1 Basic Idea

In a LOTOS specification, we specify a behavior expression of the protocol consisting of events and their temporal order. To specify the temporal order of events, we use several operators in LOTOS such as action prefix (which combines events in sequential order) as well as choice, parallel, sequential and disabling between any two sub-behavior expressions. We specify the parallel operator without synchronization gates ($||$) between two behavior expressions so that they execute their events in interleaving fashion. If we specify the parallel operator ($[[G]]$) with synchronization gates between two expressions, their events on the specified gates $G$ must be executed in synchronization with each other (called multi-rendezvous). A disabling operator ($B_1 > B_2$) enables the sub-behavior expression $B_2$ to interrupt the execution of $B_1$ by the execution of an event in $B_2$. In a LOTOS specification, we can replace any sub-behavior expression to a process instantiation. It means that we can compose the whole behavior expression as the set of structured modules (sub-processes).

In the paper, we adopt the following steps to compose hardware circuits from such structured LOTOS specifications.

1. to decompose a LOTOS behavior expression to a set of sub-behavior expressions consisting only of action prefixed sequences, choices among them and their repetition (hereafter we call such a behavior expression as the sequential behavior expression).

2. to convert each sequential behavior expression to an EFSM, and to calculate statically the multi-rendezvous table which is a set of triples (each triple consists of guard expressions, tuple of EFSMs and tuple of events for executing each multi-rendezvous).

3. to compose a hardware circuit to control multi-rendezvous among EFSMs based on the multi-rendezvous table when the EFSMs are executed in parallel as separated state machines with the same clock.

The techniques to decompose a LOTOS specification to some sequential behavior expressions with the multi-rendezvous table have been proposed [8], [11].

2.2 Converting LOTOS Specification to EFSMs

Example LOTOS Specification

Table 1 is the main part of Abracadabra protocol in LOTOS [4], which describes the behavior of a node in the file transfer protocol between two nodes. Table 1, the processes Up and Low play roles for the upper layer (i.e. user) and lower layer (i.e. network), respectively. Process Coord describes the constraints in the behavior (for cooperative work) between Up and Low. In Abracadabra protocol, once the protocol receives the connection request ($a!ConReg$) from the user, it sends the request to the other node via network ($m!Mreq!CR$). When it receives the confirmation for the connection from the other node ($m!Mnd!CR$), it informs the confirmation to the user ($a!ConEn$). Once the connection is established, it starts to send the file with the following sequences (here, note that the file is divided into multiple packets and the following sequences are repeated until the last packet is successfully transmitted): receiving the data transmission request from the user ($a!DatReq$), sending the request to the other node ($m!Mreq!DT$), receiving the acknowledgment from the other node ($m!Mnd!AK$).

To control timeout, a timer process Timer is described in the specification. If the protocol cannot receive the confirmation or the acknowledgment within a specified time after it sends the request message with starting the timer (by $!St$), Timer issues the event $!Rt$ to make the protocol resend the request message. When retransmission is repeated several times, Timer issues the event $!K1$ to stop the protocol.

Decomposing to Sequential Behavior Expressions

Basic steps of the decomposition are (1) to assign two operands separated by the parallel operators ($||$, $[[G]]$) and disabling operators ($>$) to different sequential behavior expressions, (2) to replace process instantiations to their behavior expressions. Here, note that we do not replace the process instantiations if the instantiations are appeared again while processing a sequential expression (we consider such a process instantiation as the repetition of the behavior).

First, the whole behavior expression Timer $[[t]]$ Protocol is decomposed to two processes Timer and Protocol. Protocol := (Up $||$ Low) $[a, m]$ Coord is divided in

\footnote{For simplicity, the keywords process and endproc are omitted in each process definition. And in both process definitions and their instantiations, process parameters (gate names and data values) are also omitted.}
Table 1  Simplified LOTOS specification of Abracadabra protocol.

```
specification Abra[a,m]: exit
behavior
    hide t in Timer ||[t]|| Protocol
where
    Protocol:=((Up ||| Low) ||[a,m]|| Coord)
    where
        Up:= exit [] a?x; Stick
        where Stick:= a?x; Stick [] exit
        Low:= exit [] m?x1?x2; Stick
        where Stick:= m?x1?x2; Stick [] exit
        Coord:= hide err,expt in
                Notcr ||[err,expt]| | ( (Con >> Trans) | (Dis [] Err) ) ||[a,m]|| DscCnd )
    where
        Con:= a!ConReq;t!St;m!Mreq!CR; TryCon >> a!ConConf;exit)
        [] m!Mind!CR;a!ConInd;a!ConResp;m!Mreq!CC;exit
        [] m!Mind?x[not(x==CR or x==CC)];Con
        where
            TryCon:= m!Mind?x[x==CR or x==CC]; exit
            [] m!Mind?y[y==DT or y==AK or x==CC]); TryCon
            [] t!Rt; m!Mreq!CR; TryCon
            [] t!Kl; err; stop
        Trans:= Sndr ||| Rcvr
        where
            Sndr:= a!DatReq; t!St; m!Mreq!DT; TrySnd>>Sndr
            where
                TrySnd:= m!Mind!AK;exit
                [] t!Rt; m!Mreq!DT; TrySnd
                [] t!Kl; err; stop
                [] m!Mind!AK; err; stop
        DscCnd:= a?y[not(x==DisReq or x==DisInd)]; DscCnd
        [] a?x[x==DisReq or x==DisInd]; AnyMSP
        [] m!Mind!DR;m;(a;exit [] exit)
        [] m!Mind?w[not(w==DR)]; DscCnd
        [] m!Mreq?z; DscCnd
    where
        AnyMSP:= m;AnyMSP [] exit
endspec
```

(We omit the behavior expressions for processes Timer, Dis, Err and Rcvr. The definitions for processes and events are also simplified)

three processes Up, Low and Coord. Similarly, Coord is divided in four processes Notcr, Con >> Trans, Dis [] Err and DscCnd. Processes Up, Low, Notcr, Dis [] Err and DscCnd are the sequential behavior expressions because their behavior expressions cannot be divided any more. On the other hand, in Con >> Trans, Con can be assigned to a sequential behavior expression, but Trans := Sndr ||| Rcvr can’t. In such a case, we assign Con >> Sndr and Rcvr to two sequential behavior expressions, respectively, so that the number of derived EFSMs can be reduced. When we give such an assignment, Rcvr has to be invoked just after the behavior of Con has finished in Con >> Sndr. Accordingly, we transform the expression “Con >> (Sndr ||| Rcvr)” to “hide theta in (Con >> theta; Sndr) ||[theta]|| theta; Rcvr” (such a transformation preserves the equivalence relation like weak bi-simulation).

Finally, we can get the following eight sequential behavior expressions from the specification in Table 1: Timer, Up, Low, Notcr, Con >> theta; Sndr, theta; Rcvr, Dis [] Err and DscCnd. In this way, we can derive sequential be-
behavior expressions from a structured LOTOS specification as long as it does not include infinite process instantiations such as $P := B_1 \gg P \gg B_2$ or $P := B || P$ (here, $B, B_1, B_2$ represents some behavior expressions). The details about the decomposition can be found in Ref. [8], [11].

Note that we only treat a LOTOS specification which can be decomposed into a finite number of concurrent sequential behavior expressions with multi-rendezvous using the above technique.

Converting Sequential Behavior Expressions to EFSMs

In the paper, we assume that each EFSM can have a finite number of registers, that a certain execution condition called a guard expression can be specified to each transition (i.e. edge), and that each transition can perform several substitutions of the registers in parallel.

Each sequential behavior expression can be converted to an EFSM by the following steps [2]: (1) to replace the variables, events and guard expressions to the registers, transitions and execution conditions in the EFSM, respectively, (2) to consider the whole behavior expression (initial behavior expression) and the behavior expression after executing an event, as the initial state (we denote it as $s_1$) and the state after the corresponding transition is executed, respectively.

For example, in the sequential behavior expression $Con \gg Sndr$ extracted from the specification in Table 1, the initial behavior expression is considered as the initial state $s_1$. In process $Con$, since the choice operators are specified among three events $a!ConReq, m!Mind!CR$ and $m!Mind?x_5$, these events are assigned to divergent transitions from the initial state $s_1$, and the behavior expression after the execution of each event is considered as the destination state reached by the corresponding transition (states $s_2, s_3$ and $s_1$ in Fig. 1 correspond to the destination states of the transitions, respectively). Each recursive process call (instantiation) in the sequential behavior expression is converted to the transition to the state before the behavior of the process starts. For example, in $Con \gg Sndr$, the behavior of the sub-process $TryCon$ starts at the state ($s_4$ of EFSM5 in Fig. 1) after three sequential events $a!ConReq, t!St$ and $m!Req!CR$ are executed from state $s_4$. In the behavior of $TryCon$, $TryCon$ is called recursively. Therefore, the call is mapped to the transition to state $s_4$.

We clarify the synchronization by exit operator among EFSMs, and convert disabling operators to synchronization operators as the following steps:

1. If parallel operators are specified among EFSMs like EFSM5 || EFSM6 (or EFSM5 || $G$) EFSM6, those EFSMs have to synchronize with each other by exit operator when they finish their behavior. Accordingly, we replace exit operators in the EFSMs to the special synchronization events delta, and make the EFSMs to synchronize with each other by delta. Consequently, EFSM5 || $G$ EFSM6 is replaced to EFSM5’ || $G \cup \delta$ EFSM6’ where EFSM5’ and EFSM6’ are obtained from EFSM5 and EFSM6 by replacing exit operators to delta events.

2. If disabling operators are specified among EFSMs like (EFSM5 || EFSM6) || EFSM7, EFSM5 and EFSM6 must be disabled when an event is executed in EFSM7. In order to transform disabling operators to synchronization operators, we add the transitions from all the states in EFSM5 (EFSM6) to the special state $s_1$, which represents that a disabling has occurred, and specify EFSM5 and EFSM6 to synchronize with disabling transitions in EFSM7. Here, we assume that EFSM5 and EFSM6 cannot execute the disabling transitions of EFSM7.

First we replace the disabling operator to the synchronization operator (i.e. (EFSM5 || EFSM6) || $[a, m, \exp \delta] EFSM7$). We add the special state $s_1$ to EFSM5 and EFSM6, respectively, which represents the state after disabled. Then, we add three disabling transitions $\{a!DisReq, m!Mind!Dr, \exp \}$ to every state except $s_1$ in EFSM5 and EFSM6 so that their destinations should be $s_1$ (dotted lines of EFSM5 and EFSM6 (each line corresponds to the three transitions) in Fig. 1). While a disabling has not occurred, the events on gates ‘a’ and ‘m’ in EFSM5 and EFSM6 should be able to be executed in synchronization with EFSM7. Accordingly, we add two self-loop transitions $a?x_7 [not(x_7 = \text{DisReq})]$ and $m?y_9 [not(y_9 = \text{Dr})]$ to the initial state $s_1$ of EFSM7. In addition, since EFSM7 should be able to execute the subsequent transitions on gates ‘a’ and ‘m’ and exit transitions after a disabling has occurred, self-loop transitions $a?x$ and $m?y_1 ?y_2$ and a transition exit (its destination should be the final state) are added to $s_f$ of EFSM5 and EFSM6, respectively. Finally, the obtained EFSMs with synchronization operators ((EFSM5’ || EFSM6’’) || $[a, m, \exp \delta] EFSM7’$) are equivalent to the original ones in terms of weakly bi-simulation.

Figure 1 depicts the EFSMs converted from the sequential behavior expressions Up, Low, Con >> theta; Sndr, theta; Rovr and DscCnd.

3. Behavior of EFSMs with Multi-Rendezvous

This section provides the definition of the behavior of EFSMs with multi-rendezvous (hereafter, we call them as synchronous EFSMs) derived in Sect. 2, and how those EFSMs work in cooperation with each other.

3.1 Definition of Behavior of Synchronous EFSMs

Let $E = \{efsm_1, \ldots, efsm_n\}$ be the set of EFSMs. If $E \subseteq \Sigma$ is specified to synchronize with each other on a gate $g$, we describe it as Rend($E, g$). We call each Rend($E, g$) as the synchronization indication. Let $R$ be the set of all synchronization indications given on $E$. We can give two different indications Rend($E_1, g_1$) and Rend($E_2, g_2$) such that $E_1 \cap E_2 \neq \emptyset$.
Table 2 Synchronization condition.

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
<th>condition</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a!E_i$</td>
<td>$a!E_j$</td>
<td>$val(E_i) = val(E_j)$</td>
<td>value matching</td>
</tr>
<tr>
<td>$a!E_i$</td>
<td>$a?x: t$</td>
<td>$val(E_i) \in \text{domain}(t)$</td>
<td>value passing</td>
</tr>
<tr>
<td>$a?x: t$</td>
<td>$a?y: u$</td>
<td>$t = u$</td>
<td>value generation</td>
</tr>
</tbody>
</table>

(val($E$) is the normal form of the expression $E$, domain($t$) is the domain of the sort $t$)

Now, suppose that $\text{Rend}(E, g) \in \mathcal{R},$ $E \subseteq \mathcal{E}, E = \{efsm_1, \ldots, efsm_m\}$ are given. Let $ev(s_i)$ be the set of transitions from a state $s_i$ of each EFSM. If the following conditions hold for the tuple of events $(e_1, \ldots, e_m)$, we say that the tuple of states $(s_1, \ldots, s_m)$ is ready-to-rendezvous, and that the tuple of events $(e_1, \ldots, e_m)$ is the synchronization tuple (we also call each event in the synchronization tuple as a synchronous event).

- there is a tuple of events $(e_1, \ldots, e_m)$ such that $e_i \in ev(s_i)$ ($1 \leq i \leq m$)
- any pair of events $e_i$ and $e_j$ ($1 \leq i, j \leq m, i \neq j$) in the tuple satisfies the synchronization condition in Table 2
- the execution conditions of $e_1, \ldots, e_m$ are all true

After a synchronization tuple is executed, the value of an output event (e.g. $a!\text{ConReg}$) in the tuple is assigned to all input variables of events in the tuple. Here, we assume that the input value $\text{Ext}$ at the external gate is assigned to all input variables when all events in the tuple have no output events.

Let gate($e$) represent the gate name of the event $e$. We call the event $e$ as an asynchronous event if gate($e$) $\neq g$ for every synchronization indication $\text{Rend}(E, g) \in \mathcal{R}$.

Using the above notions, we define the behavior of each EFSM in the synchronous EFSMs as follows:

- only an asynchronous event can be executed while the current tuple of states of EFSMs is not ready-to-rendezvous.
- either an asynchronous event or a synchronous event can be executed when the current tuple of states is ready-to-rendezvous.
- if the EFSM can execute asynchronous events and/or synchronous events, one of them must be selected to execute.

If different two synchronization indications $\text{Rend}(E_1, g_1), \text{Rend}(E_2, g_2) \in \mathcal{R}$ satisfy $E_1 \cap E_2 = \emptyset$, they must not be executed simultaneously (mutual exclusiveness).

3.2 Calculation of Multi-Rendezvous Table

Only synchronization operators are specified among the EFSMs derived in Sect. 2 as shown in Table 3.

From the syntax tree of the operators among EFSMs and gate names used in each EFSM, we can get the synchronization indications defined in the previous section. For example, from the syntax tree in Table 3 and the contents of EFSMs, the following synchronization indications on gates ‘a’ and ‘m’ can be derived.

\[ \text{Rend}(\{\text{EFSM2, EFSM5, EFSM7, EFSM8}\}, a). \]
\[ \text{Rend}(\{\text{EFSM2, EFSM6, EFSM7, EFSM8}\}, m). \]
The finite set of all the synchronization tuples with their execution conditions and the combination of EFSMs is statically determined with given EFSMs and synchronization indications. We call such information as multi-rendezvous table.

We can calculate the multi-rendezvous table statically from the given EFSMs and synchronization indications \( \mathcal{R} \), as the following steps:

For each synchronization indication \( \text{Rend}(E, g) \in \mathcal{R}, E = \{\text{efsm}_1, \cdots, \text{efsm}_m\} \),

- to extract all events such that \( \text{gate}(e) = g \), for each \( \text{efsm}_i \in E \) (let \( \text{sync} \_\text{ev}(\text{efsm}_i, g) \) be the extracted events for \( \text{efsm}_i \)).
- for each \( e_i \in \text{sync} \_\text{ev}(\text{efsm}_i, g) \), to calculate a tuple \( (e_1, \cdots, e_m) \) which satisfies the synchronization conditions in Table 2.

In general, if we calculate the multi-rendezvous table simply as the combinations of events in EFSMs, the number of synchronization tuples will be quite numerous (e.g. if each of \( n \) EFSMs has \( m \) events on gate \( g \) and all EFSMs are specified to synchronize on \( g \), the number of the synchronization tuples will be \( m^n \)).

Accordingly, to reduce the number of elements in the multi-rendezvous table, we merge the synchronization tuples using the following technique:

(1) to calculate the set of output values \( \text{OV} \) where each value is assigned to undefined variables by the synchronization, from all the synchronization tuples calculated among \( E = \{\text{efsm}_1, \cdots, \text{efsm}_m\} \).

(2) for each \( \text{efsm}_i \in E \) and each \( v \in \text{OV} \), to calculate the set of events \( \mathcal{A}_i \) such that each event satisfies the synchronization conditions in Table 2 with \( v \).

(3) to construct the multi-rendezvous table such that each member of the table should be a tuple \( (E, (\mathcal{A}_1, \cdots, \mathcal{A}_m)) \).

We call each member of the multi-rendezvous table as the rendezvous information. For each rendezvous information \( r = (E, (\mathcal{A}_1, \cdots, \mathcal{A}_m)) \), we represent each element of the set \( \mathcal{A}_i \) as the triple \( (e, p, I) \). Here, \( e \) is the event name consisting of a gate name and input/output parameters, \( p \) is a guard expression, and \( I \) is the set of substitutions to assign some values to the registers. For each \( e_i \in A_i \), the tuple \( (e_1, \cdots, e_m) \) satisfies the synchronization conditions in Table 2. For example, given EFSM1 := \( (\text{efsm}_1, \cdots, \text{efsm}_m) \), \( (A_1, \cdots, A_m) \), if each \( \text{efsm}_i \) is in its state to execute the event \( e_i \in A_i \) and the execution condition of \( e_i \) is true, then the synchronization tuple \( (e_1, \cdots, e_m) \) can be executed. Whether some synchronization tuples \( (e_1, \cdots, e_m) \) can be executed or not, is decided by checking the existence of rendezvous information which contains such a synchronization tuple.

In the initial tuple of states \( (s_1, s_1, s_1, s_1) \) of EFSM2, EFSM5, EFSM7 and EFSM8, these EFSMs can execute events \( a?x_2, a!\text{ConReq}, a?x_7 \text{ not}(x_7 = \text{DisReq}) \) \( a?x_8 \), respectively, and the execution conditions for the events are true. Therefore, in the states, the tuple \( (a?x_2, a!\text{ConReq}, a?x_7, a?x_8) \) can be executed by the rendezvous information (1). When the tuple \( (a?x_2, a!\text{ConReq}, a?x_7, a?x_8) \) is executed, the output value \( \text{ConReq} \) is assigned to the undefined variables (registers) \( x_2, x_7 \) and \( x_8 \), and the tuple of the current states is changed to \( (s_2, s_2, s_1, s_1) \).

In some tuple of the states, there may be several synchronization tuples to be executable simultaneously. For example, in Fig. 1, when the tuple of the current states is \( (s_1, s_1, s_1, s_1) \) for EFSM2, EFSM3, EFSM5, EFSM7 and EFSM8, two synchronization tuples \( (m?x_{31} \& x_{32}, m!\text{Mind} \& x_5, m?y_{71} \& y_{72}, m!\text{Mind} \& w_{39}) \) and \( (m?x_{31} \& x_{32}, m!\text{Mind} \& c_{39}, m?y_{71} \& y_{72}, m!\text{Mind} \& w_{39}) \) can be executed among EFSM3, EFSM5, EFSM7 and EFSM8 by the rendezvous information (3) and (4) as well as \( (a?x_2, ...
a \mid \text{ConReq}, a \equiv x_7, a \equiv x_8) among EFSM2, EFSM5, EFSM7 and EFSM8 by the rendezvous information (1). If (a \equiv x_2, a \mid \text{ConReq}, a \equiv x_7, a \equiv x_8) is executed, the synchronization tuples among EFSM3, EFSM5, EFSM7 and EFSM8 must not be executed any more.

As explained above, when several mutually exclusive synchronization tuples become executable, one of them must be selected under the consensus of the related EFSMs.

4. Conversion from EFSMs with Multi-Rendezvous into Circuit

In this section, we give a technique to convert given EFSMs with a multi-rendezvous table into a synchronous sequential circuit where the submodules corresponding to the EFSMs are running synchronously using the same clock.

4.1 Restrictions for Multi-Rendezvous Table

For simplifying the conversion, we give the following restrictions for each rendezvous information in the multi-rendezvous table.

1. If a synchronous event set for an EFSM has some events, those events must be either all input events (\text{By}) or all output events (\text{By})
2. If an EFSM has a state which has some outgoing transitions with different kind of output events, the output events must not be contained in the same rendezvous information.
3. The number of EFSMs whose synchronous event set is a set of output events is at most one.

Generality is not lost by the above restrictions for the following reasons: Restriction 1 and Restriction 2 simply make some rule for merging the synchronous events to each rendezvous information; for Restriction 3, if there is a multi-rendezvous which includes multiple output events like a \equiv x \mid [a] \mid a y \mid [a] \mid a z, we can transform the expression to an equivalent expression which includes only one output event like a \equiv x \mid [a] \mid a w \mid [w = y] \mid [a] \mid a z.

4.2 Multi-rendezvous Executability Check Part

Given EFSMs and a multi-rendezvous table, the following work is required for making EFSMs to behave with the multi-rendezvous table: (1) checking whether there exist executable synchronization tuples for each rendezvous information at each time; (2) selecting a synchronization tuple set to execute with considering avoidance of conflicting mutual exclusive synchronization tuples; (3) transferring the required data among appropriate EFSMs.

For the above (1), every EFSM in each rendezvous information must check whether some events in its synchronous event set are executable at the current state. So, for every \text{eFSM}_j, we provide a circuit which generates an output signal (i.e. \text{e}_j \circ \omega k) which means that one of the corresponding synchronous event set in \text{eFSM}_j is executable for the rendezvous information \text{r}_i. Consequently, for the rendezvous information \text{r}_i, there exist some executable synchronization tuples if and only if \text{e}_j \circ \omega k of all EFSMs in the rendezvous information are \text{true}.

For the above (2), we provide the multi-rendezvous conflict avoidance part. This part selects a synchronization tuple set to execute by evaluating the outputs \text{e}_j \circ \omega k and outputs the result (i.e. \text{r}_i \ast \text{en}) to all EFSMs in the rendezvous information \text{r}_i. In general, any hardware implementation cannot perform non-determinism in a LOTOS specification precisely. But, in practice, there exist acceptable deterministic selecting policies (e.g. deciding by priorities or use of a pseudo-random value which can be implement by a counter/timer). In this paper, we present a selecting (or conflict avoiding) method based on priorities of rendezvous information as a reasonable solution (described later).

For the above (3), we provide the data transfer part. Data
Synchronizing event sets

In addition to the rendezvous information, EFSMs with output events are determined statically by the restrictions described in Sect. 4.1. And in data transfer of each rendezvous information, EFSMS with input events and an output event will not need any data transfer. Each EFSM with output events outputs an appropriate value for the current state to the path. Each EFSM with output events outputs the value of the path and calculates the execution condition with the value or assigns the value to its own variables.

The configuration of the circuit including EFSMs is shown in Fig. 2. Furthermore, the circuit for each EFSM can be implemented easily by well-known techniques [10].

4.3 Method of Selecting A Synchronization Tuple Set by Priority

As mentioned earlier, the multi-rendezvous conflict avoidance part selects a synchronization tuple set to execute by evaluating the outputs $e_{i,ok}$ from all EFSMs and outputs $r_{i,em}$ of each rendezvous information $r_i$.

Synchronization tuples of rendezvous information $r_1$ and $r_2$ can conflict if and only if there exists the case that for a state set $(s_1, \ldots, s_p)$ of the common EFSMs in both rendezvous information, the synchronous events of $r_1$ and $r_2$ are both executable at every state $s_k (1 \leq k \leq p)$. For all combinations of two rendezvous information, we can determine statically whether they can conflict or not by checking given EFSMs and multi-rendezvous table.

In addition, for the execution of a synchronization tuple of rendezvous information $r_i$, all synchronization tuples of rendezvous information with higher priorities than $r_i$ which conflict with $r_i$ must not be executable.

Consequently, we provide the following part for each rendezvous information $r_i$. The inputs of the part are (1) the outputs $e_{i,ok}$ from all $efsm_j$ in the $r_i$, each of which means whether any synchronous events are executable in $efsm_j$ and (2) $pri_i$ which means whether a synchronization tuple of $r_i$ has the right to execute or not. The output of the part is $r_{i,em}$ which means whether any synchronization tuples of $r_i$ are enabled to execute or not. The output value from $r_{i,em}$ is the following:

$$r_{i,em} = e_{i,ok}^1 \land \cdots \land e_{i,ok}^m \land pri_i$$

Here, $\{r_{i,1}, \ldots, r_{i,n}\}$ are the rendezvous information with higher priorities than $r_i$ which conflict with $r_i$.

4.4 An Example of Derived Circuit

In this section, we explain how we can derive the circuit in Fig. 4 which is obtained by applying the proposed method to the EFSMs in Fig. 3 and multi-rendezvous table in Table 6.

At the initial state $(s_1, s_1)$, first, EFSM1 calculates the output value $a_{1,ok}$ for the rendezvous information $r_1$ by following. As EFSM1’s current state is $s_1$, EFSM1 calculates the execution condition $p(x_1) \lor q(x_2)$ for the events $a?x_1 [p(x_1)]$ and $a?x_2 [q(x_2)]$ which are transitions from state $s_1$ in the event set $a_1$, respectively. Furthermore, since $x_1$ and $x_2$ are external values, EFSM1 uses the value from the data path $D_1$ (the data path for the rendezvous execution $r_1$) as the values of $x_1$ and $x_2$. So, it outputs the value of $p(D_1) \lor q(D_1)$ to $a_{1,ok}$. On the other hand, as EFSM2’s current state is $s_1$, EFSM2 can execute the events $a_{1,1}$ in $a_2$ for the rendezvous information $r_1$. So it outputs true to $a_{2,ok}$. In addition, since $a_{1,1}$ is an output event, it outputs the value 1 to the data path $D_1$. For other rendezvous information, EFSMs behave in the same way.

If $a_{1,ok}$ and $a_{3,ok}$ are both true, $r_1$ and $r_2$ are conflict.
We have constructed the circuit for the Abracadabra protocol in LOTOS [4] with our method and have evaluated the conflict avoidance part of concurrent EFSMs. This experiment has aimed at showing that the control part of the circuit does not include the modules which calculate the execution conditions for the synchronous events in each EFSM for the following reasons:

(1) because the contents of calculation for execution conditions depend on how to implement the data types and values on the target circuit, (2) designers need not take care of the decrease of the performance (or the maximum speed of the clock) which is caused by the conflict avoidance part if we would show that the processing time (or the number of the steps of the logical gates) in the conflict avoidance part was reasonably small.

As mentioned in Sect. 2, we have implemented Abracadabra protocol as eight EFSMs within this experiment. The number of rendezvous information is 85, and the maximum number of the steps for avoiding the conflict of exclusively executable rendezvous information is seven. The size of the conflict avoidance part grows linearly in the number of rendezvous information since each rendezvous information has its own module. The time spent for selecting a synchronization tuple set grows linearly in the maximum number of rendezvous information which may be conflict each other.

We have designed a circuit by our proposed method described in the previous section, and have obtained the circuit by a hardware synthesis system PARTHENON [9] which has been developed by NTT. The size of the whole circuit is about 5000 gates. This value depends on the size of implemented data and the above value is calculated for eight-bit data. The conflict avoidance part has about 300 gates and the maximum number of the steps of the logical gates is six. We think this result is reasonable in practice.

5. Experimental results and discussion

We have constructed the circuit for the Abracadabra protocol in LOTOS [4] with our method and have evaluated the conflict avoidance part of concurrent EFSMs. This experiment has aimed at showing that the control part of the circuit constructed by our method is reasonably small and fast. In this evaluation, the constructed control part does not include the modules which calculate the execution conditions for the synchronous events in each EFSM for the following reasons: (1) because the contents of calculation for execution conditions depend on how to implement the data types and values on the target circuit, the criteria for the circuits also depends on the target circuit, (2) designers need not take care of the decrease of the performance (or the maximum speed of the clock) which is caused by the conflict avoidance part if we would show that the processing time (or the number of the steps of the logical gates) in the conflict avoidance part was reasonably small.

As mentioned in Sect. 2, we have implemented Abracadabra protocol as eight EFSMs within this experiment. The number of rendezvous information is 85, and the maximum number of the steps for avoiding the conflict of exclusively executable rendezvous information is seven. The size of the conflict avoidance part grows linearly in the number of rendezvous information since each rendezvous information has its own module. The time spent for selecting a synchronization tuple set grows linearly in the maximum number of rendezvous information which may be conflict each other.

We have designed a circuit by our proposed method described in the previous section, and have obtained the circuit by a hardware synthesis system PARTHENON [9] which has been developed by NTT. The size of the whole circuit is about 5000 gates. This value depends on the size of implemented data and the above value is calculated for eight-bit data. The conflict avoidance part has about 300 gates and the maximum number of the steps of the logical gates is six. We think this result is reasonable in practice.

6. Conclusions

In this paper, we have proposed a method to derive a register transfer level circuit from a LOTOS specification consisting of EFSMs with multi-rendezvous, and have evaluated the size of the circuit and the number of its gate steps for avoiding conflict of multi-rendezvous.

As future work, we plan to develop a synthesis system which implements our method, and to make sure that our method is effective in practice by applying our method to more practical examples.

References

Akira KITAJIMA received the M.E. degree in Information and Computer Sciences from Osaka University in 1995. He is now a Ph.D. student of Department of Information and Computer Sciences, Osaka University. His research interests include synthesis and formal verification of hardware. He is a member of IPS of Japan.

Keiichi YASUMOTO received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Osaka, Japan, in 1991, 1993 and 1996, respectively. He joined the faculty of Shiga University in 1995. Since 1996 he has been an Assistant Professor. In 1997, he is a Visiting Researcher of Dept. I.R.O. at University of Montreal, Canada. His current research interests include design and implementation of distributed systems. He is a member of IPS of Japan.

Teruo HIGASHINO received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Osaka, Japan, in 1979, 1981 and 1984, respectively. He joined the faculty of Osaka University in 1984. Since 1991 he has been an Associate Professor in the Department of Information and Computer Sciences at Osaka University. In 1990 and 1994, he was a Visiting Researcher of Dept. I.R.O. at University of Montreal, Canada. His current research interests include design and analysis of distributed systems, specification and verification of communication protocols, and formal approach of program design. He is a member of IEEE, ACM, IEICE of Japan, and IPS of Japan.

Kenichi TANIGUCHI received the B.E. and M.E. degrees in electronics engineering and Ph.D. degree in control engineering from Osaka University, Osaka, Japan, in 1965, 1967 and 1970, respectively. He is a Professor of the Department of Information and Computer Sciences, Osaka University. His research interests include design method for distributed systems and communication protocols and algebraic methods for software development and hardware design and verification.