PAPER

Deriving Concurrent Synchronous EFSMs from Protocol Specifications in LOTOS

Akira KITAJIMA\(^1\), Keiichi YASUMOTO\(^\dagger\), Nonmembers, Teruo HIGASHINO\(^1\), and Kenichi TANIGUCHI\(^1\), Members

SUMMARY

In this paper, we propose an algorithm to convert a given structured LOTOS specification into an equivalent flattened model called synchronous EFSMs. The synchronous EFSMs model is an execution model for communication protocols and distributed systems where each system consists of concurrent EFSMs and a finite set of multi-rendezvous indications among their subsets. The EFSMs can be derived from a specification in a sub-class of LOTOS and its implementation becomes simpler than the straightforward implementation of the original LOTOS specification because the synchronization among the processes in the model does not have any child-parent relationships, which can make the synchronization mechanism much more complex. Some experimental results are reported to show the advantage of synchronous EFSMs in terms of execution efficiency.

key words: LOTOS, synchronous EFSMs, transformation, multi-rendezvous, implementation

1. Introduction

For effective design and development of complicated communication protocols and distributed systems, formal description techniques (FDTs) are increasingly getting important. An FDT LOTOS developed within ISO has some useful features to specify concurrent systems structurally and simply.

In general, however, a complicated control mechanism is required to implement such structured concurrent processes in a LOTOS specification since multiple processes can be dynamically invoked or terminated and they interact with each other by synchronization, interruption and so on. So, a concrete implementation needs a mechanism to invoke/terminate processes and to keep their latest executional dependence relation (i.e. structured operators specified among processes) like in [7] and [9], which may bring the implementation overhead. One effective solution may be to convert given structured concurrent processes into equivalent flattened concurrent processes. For example, [3] and [1] convert a very restricted class of LOTOS specifications into equivalent automata and their synchronization with ad-hoc techniques, and execute them rather efficiently. For general LOTOS specifications, [8] has proposed an algorithm to statically derive simple parallel processes and information on their synchronization. However, the technique requires the complete reachability analysis among all parallel processes which needs the time proportional to the product of the numbers of events (transitions) in the concurrent processes (therefore, not practical).

In this paper, we propose an algorithm to convert a given LOTOS specification into an equivalent synchronous EFSMs model [6] where we only give a restriction that, in the LOTOS specification, the number of concurrent processes invoked in parallel must be finite. Synchronous EFSMs model consists of the fixed number of EFSMs running in parallel and a table representing all possible synchronization called multi-rendezvous[4] among them. In this model, since there is no child-parent relationships about the synchronization among concurrent EFSMs, the model can be simply implemented. In the proposed algorithm, first we transform a LOTOS specification to the parallel composition of sequential behavior expressions (SBEs), each of which consists of event sequences, their choice and iteration with the corresponding behavior expression. Each SBE can be converted to an EFSM. We get a multi-rendezvous table statically without reachability analysis from the information about transitions in each EFSM and synchronization operators specified among the EFSMs.

The rest of this paper is organized as follows. In Sect. 2, we introduce LOTOS with a typical concurrent system, and give the definition of synchronous EFSMs. An algorithm to derive the synchronous EFSMs from a LOTOS specification is given in Sect. 3. In Sect. 4, we report some evaluation results to show the synchronous EFSMs can be implemented efficiently.

2. LOTOS and synchronous EFSMs

2.1 LOTOS and its application

In this section, we briefly introduce the syntax, semantics and some features of LOTOS as well as an example of protocol specification. For the formal definition of LOTOS, see [4].

2.1.1 Syntax and semantics

In a LOTOS specification, we specify a behavior expression consisting of events and their temporal order. To specify the order, we use the following LOTOS operators. Action prefix (a; B) combines events sequentially with ‘;’.
Here we allow should be carried out concurrently with the data reception. To calculate the check sum is longer than the average period between two consecutive data receptions, such calculation is automatically selected by the property of multi-rendezvous. We can adjust the value of \( M \) depending on the time for calculation and the frequency of data receptions at gate \( g \).

Choice \((B_3 \parallel B_2)\) selects one of either \( B_1 \) or \( B_2 \). Disabling \((B_1 \triangleright B_2)\) allows \( B_2 \) to interrupt the execution of \( B_1 \) by execution of an event in \( B_2 \). Sequential composition \((B_1 \triangleright\triangleright B_2)\) allows \( B_2 \) to proceed when \( B_1 \) finishes successfully with exit operator. Parallel/Synchronization \((B_1 \parallel [G]; B_2)\) specifies that pairs of events of \( B_1 \) and \( B_2 \) on \( G \) must be executed in synchronization with each other (if \( G \) is empty (i.e. [ ]), no events need to be synchronized). When several expressions are combined with parallel operators like \( B_1 \parallel [G_1] \cdots [G_{n-1}] \parallel B_n \), the events on the common gates \( G_1 \cdots G_{n-1} \) need be executed synchronously (called multi-rendezvous). In general, we define that a multi-rendezvous specified among some processes on a gate (e.g. \( g \)) can be executed only if all of these processes can execute their events on \( g \) and any pair of the events satisfies the synchronization condition in Table 1. When the rendezvous is executed, the output values are assigned to the undefined variables. Here, note that if either \( B_1 \) or \( B_2 \) in \( B_1 \parallel [G]; B_2 \) contains multiple events on a gate \( g \) (\( g \in G \)), there will be multiple combinations of synchronizing events on \( g \).

### 2.1.2 Specification example

Let us describe an example protocol in LOTOS as a typical concurrent system. In usual protocols, in order to send data via unreliable medium, we need to calculate the check sum of each transferred data. The sequential number may also be attached to each data packet. Let us suppose the protocol in Fig. 1, which receives the data successively from its upper layer (i.e. from its service user) via gate \( a \) and sends the data packet with its check sum (parity bit) and the sequential number to the lower layer (i.e. towards the other node) via gate \( m \). For the sake of simplicity, we assume the check sum for each \( N \) bit data is calculated in \( N \) steps (see CkSum in Fig. 1). If the time to calculate the check sum is longer than the average period between two consecutive data receptions, such calculation should be carried out concurrently with the data reception. Here we allow \( M \) processes to calculate the check sums concurrently. The whole protocol can be described with multi-rendezvous as shown in Fig. 2.

In Fig. 2, process AsgnTsk produces \( M \) concurrent processes of CkSum to calculate the check sums. Process Upper and CkSum receive data \( x \) at the same time with multi-rendezvous at gate \( a \). Here, it is important that one of the processes of CkSum ready to calculate the check sum is automatically selected by the property of multi-rendezvous. We can adjust the value of \( M \) depending on the time for calculation and the frequency of data receptions at gate \( g \).

Table 1  Synchronization condition.

<table>
<thead>
<tr>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>condition</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g[E_1] )</td>
<td>( g[E_2] )</td>
<td>( val(E_1) = val(E_2) )</td>
<td>value matching</td>
</tr>
<tr>
<td>( g?x : t )</td>
<td>( g?y : u )</td>
<td>( t = u )</td>
<td>value passing</td>
</tr>
</tbody>
</table>

\((val(E))\) is the normal form of the expression \( E \), \( \text{domain}(t) \) is the domain of the sort \( t \). \( Ext \) is the value input at gate \( g \).
specification Protocol[a,m](sn:int): noexit
behavior
  hide g in
  Upper[a,g](0) || [a,g] AsgnTsk[a,g](0)
  [g] Lower[m,g](SN, 0)
where
  process Upper[a,g](id:int):noexit :=
    a?x:data !id; Upper[a,g](id+1)
  endproc
  process AsgnTsk[a,g](n:int): noexit :=
    a?x:data ?id:int; CkSum[a,g](x,0,0, id)
    [n < M]-> AsgnTsk[a,g](n+1)
  where
  process CkSum[a,g](x: data, n:int, sum: bit, id: int):
    noexit :=
    [n < N]-> CkSum[a,g](x,n+1,sum+test(x,n), id)
    [n== N]-> g!x!sum!id; a?new_x:data ?new_id:int;
    CkSum[a,g](new_x,0,0, new_id)
  endproc
  endproc
  process Lower[m,g](sn:int, id:int): noexit :=
    g?x:data ?sum:bit !id; m!packet(sn,x,sum);
    Lower[m,g](sn+1, id + 1)
  endproc
endspec
(here, SN denotes the initial sequential number. data and bit mean the sorts for data of N bits and of 1 bit, respectively. test of $p$, $q$, $L$-th bit of $p$)

Fig. 2 A LOTOS specification of an concurrent system.

Table 2 An example of LOTOS specification.

spec. := (P || Q || R) | (S || T)
where P := (d3: exit) || (P1 || P2)
Q := b; Q
R := a?x: int[x > 0]: R || b R
S := (S1 || S2) || S3
T := a; stop
(Here, the declarations of processes P1, P2, S1, S2 and S3 are omitted.)

Fig. 3 Executional dependence relation and its dynamic change.

dependence relation should be kept like in [7] and [9], although such a mechanism may bring the implementation overhead. Therefore, in this paper, we convert a given LOTOS specification into equivalent flattened model defined in the following section.

2.2 Definition of synchronous EFSMs

Synchronous EFSMs are given as a set of EFSMs \( \{ \text{efsm}_1, \ldots, \text{efsm}_n \} \) and a multi-rendezvous table \( \mathcal{R} \) which can represent all possible multi-rendezvous instances by a set of rendezvous indications. We suppose 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 rendezvous indication is given for a combination of synchronizing EFSMs on a gate (e.g. $g$) by \( \langle (E_1, \ldots, E_m), (A_1, \ldots, A_m) \rangle \) where \( (E_1, \ldots, E_m) \) is a tuple of synchronizing EFSM names, and each \( A_k \) is the synchronous transition set which contains transitions executed in \( E_i \) for the rendezvous. We represent each element of \( A_i \) as the triple $(e, p, I)$. Here, $e$ is the transition 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 $(e_1, \ldots, e_m)$ has a common gate $g$ and satisfies the synchronization conditions in Table 1. In the tuple $(e_1, \ldots, e_m)$, only a transition is an output transition and the others are input transitions. Thus, it is much simpler to implement each rendezvous indication than LOTOS multi-rendezvous since we can easily know in each rendezvous indication what EFSM outputs the value and what EFSMs expect the value as their inputs, and we don’t have to check the synchronization condition (see [6] for detail).

In synchronous EFSMs, a transition tuple is executed in synchronization with each other if and only if there is a ren-
dezvous indication \( \langle (E_1, \cdots, E_m), (A_1, \cdots, A_m) \rangle \) such that \( E_1, \cdots, E_m \) have at least one transition in respective \( A_1, \cdots, A_m \) in the current state. Transitions which are not included in any rendezvous indications can be executed any time.

3. Deriving synchronous EFSMs from a LOTOS specification

We convert a LOTOS specification to synchronous EFSMs as the following steps: (1) to transform the main behavior expression into the parallel composition among sequential behavior expressions (SBEs); (2) to convert each SBE to an EFSM; (3) to calculate a rendezvous table from the information about events in each EFSM and parallel operators among them.

In the following, first we give some preliminaries before giving the conversion algorithm, then we give algorithms for each step.

3.1 Preliminaries

In this paper, we consider any LOTOS specification represented in the class of Table 3 as long as it does not produce an infinite number of concurrent processes. Therefore, we do not treat unguarded recursive process instantiations such as: \( P := (B_1 >> P >> B_2) \parallel \text{exit} \) or \( P := B \circ P \) (\( \circ \in \{ [ ], [ ]^*, [ ]^+ \} \)). Indirect unguarded process instantiations (e.g., \( P := B \circ Q \) where \( Q \) calls \( P \)) are not allowed, either. However, we treat guarded recursive process instantiations for parallel and sequential operators (e.g., \( P(x) := B \parallel \{ [x < 100] \rightarrow P(x + 1) \} \)) as long as the guard expression for the recursive instantiations can be calculated statically.

Let \( \#Par(B) \) be the function that represents how many parallel processes can be activated at the same time in \( B \) (See Fig. 4 for the formal definition of \( \#Par(B) \)).

We say a behavior expression \( B \) is the sequential behavior expression (SBE) if \( \#Par(B) = 1 \) and \( B \) includes only action prefixed sequences, choices among them and iteration. If an SBE is not the sub-expression of any other SBEs in a given whole expression, we say that the SBE is maximal. Intuitively, the maximal SBE corresponds to the sub-behavior expression separated by parallel operators.

Table 3  Target class of LOTOS.

| \( B_0 \) | := \text{hide} G \in B_1 | B_1 |
| \( B_1 \) | := B_1 >> B_2 | B_2 |
| \( B_2 \) | := B_2 \parallel B_3 | B_3 |
| \( B_3 \) | := B_3 \parallel B_4 | B_3 \parallel B_4 | B_3 \parallel B_4 |
| \( B_4 \) | := B_4 \parallel B_5 | B_5 |
| \( B_5 \) | := [\text{boolean exp}] \rightarrow B_6 | B_6 |
| \( B_6 \) | := \text{event} B | \text{stop} | \text{exit}(B_1) | P[G] (\text{el}) |
| \( \text{event} \) | := \text{gate} | \text{gate val} \parallel \text{gate val}[\text{boolean exp}] |
| \( G \) | := \text{gate} | \text{gate} G |
| \( \text{val} \) | := ?\text{varname} : \text{sort val} | \text{exp val} |
| \( \text{exp} \) | := ( (\text{every expression written in ACT ONE} *) |
| \( \text{el} \) | := \text{exp} , \text{el} |

For example, for an expression \( a; g; b \parallel [g] \parallel c; g; d \), sub-expressions \( a, g; a \parallel g \) and \( a; g \parallel b \) are all SBEs and \( a; g; b \) is the maximal SBE in those subexpressions.

We use \( St(B) \) to refer to the set of the first executable events in \( B \). We denote the behavior expression \( B_1 \parallel [G_1] \parallel \cdots \parallel [G_{m-1}] \parallel B_m \) as \( \prod_{i=1}^{m} B_i \), and \( B_1 \parallel \cdots \parallel B_l \) as \( \sum_{i=1}^{l} B_i \).

3.2 To get maximal sequential behavior expressions

Let \( B_{main} \) be the main behavior expression of a given LOTOS specification. Here, we transform \( B_{main} \) to the parallel composition among maximal SBEs. As described later, each maximal SBE can be implemented as an EFSM on hardware or software. The number of derived EFSMs (or maximal SBEs) seems reasonable in the implementation when we extract SBEs separated with parallel operators. If several parallel processes are implemented as a single EFSM, the number of global states will grow very large as the number of parallel processes or the number of their states increase. On the other hand, as a specification is implemented as quite many EFSMs, more complicated control mechanism is required in general.

We apply the following operations recursively to each sub-behavior expression of \( B_{main} \) for the transformation of \( B_{main} \):

1. replacing each process instantiation with its behavior expression unless the instantiation appears as a tail recursion in its behavior expression (i.e. \( P := B \parallel P \) )
2. transforming each action prefixed sequence whose parallel degree is more than one, into the parallel composition among sequential behavior expressions (SBEs).
3. transforming each choice/disabling/sequential composition among sub-behavior expressions into either an SBE or a parallel composition among SBEs.

For (1), we replace each process instantiation \( P[G] (V) \) appeared in its behavior expression \( B_P \) as a tail recursion, with the iteration of \( B_P \) by introducing \( \text{label}(P[G]) \): and \( \text{goto}(P[G], X:=V) \). Although other process instantiations are
Algorithm $Trans(B, \ pset)$

begin
if $(B = a_1; \ldots; a_i; B^'; B^' \neq a'; B'^\prime)$ then
if $(\#Par(B') \geq 2)$ then
$Trans(B', \ pset)$
$TransAct(B)$
endif
else if $(B = [guard] \rightarrow B')$ then
if (guard is true or cannot be calculated statically) then
$Trans(B')$
endif
else if $(B = \sum B_i)$ then
for each $i$, $Trans(B_i, \ pset)$
$TransChoice(B)$
else if $(B = B1 || [G] || B2)$ then
$Trans(B1, \emptyset); Trans(B2, \emptyset)$
else if $(B = B1 \trianglerightl B2)$ then
$Trans(B1, \ pset); Trans(B2, \ pset); TransDis(B)$
else if $(B = B_i >> \cdots >> B_n)$ then
for each $i < k$, $Trans(B_i, \emptyset)$
$Trans(B_k, \ pset)$
$TransSeq(B)$
else if $(B = P[G](V))$ then
if $(P[G] \in \ pset)$ then
replace $P[G](V)$ with $goto(P[G], X := V)$
else
replace $P[G](V)$ with $label(P[G])$: let $X := V$ in $B_{P[G]}$
$Trans(B_{P[G]} \ pset \cup \{P[G]\})$
endif
endif
end
egin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig6.png}
\caption{Assignment and resulting SBEs in a choice expression.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5.png}
\caption{Algorithm $Trans(B, \ pset)$}
\end{figure}

replaced with their behavior expressions even if they are recursive processes, they are not infinitely instantiated by the restriction in Sect. 3.1.

We show the transformation algorithm $Trans$ in Fig. 5. Here, $pset$ is the set of process instantiations already replaced with their behavior expressions.

$TransAct$: transformation of action prefixed sequence

For $TransAct$, $a_1; \cdots; a_i; (B_1 \mid [G_1] \mid \cdots \mid [G_{n-1}] \mid B_n)$ $(def. = B_{act})$ is given. Here, each $B_i$ is an SBE because $Trans$ transforms each sub-behavior expression which is not an SBE to the parallel composition of SBEs recursively. In $B_{act}$, transforms each sub-behavior which is not an SBE to the parallel composition of SBEs recursively. In $B_{act}$, each $B_i$ should be activated after $a_i$ is executed. Accordingly, we assign the event sequence $a_1; \cdots; a_i$ and the behavior expressions $B_1, \cdots, B_n$ to different SBEs, respectively. Then, we introduce an internal signal $\theta_k$ where $\theta_k$ is a unique gate name in the whole expression before this transformation so that the SBE for $a_1; \cdots; a_i$ sends the signal $\theta_k$ to other SBEs after the execution of $a_i$, and that the SBEs for $B_1, \cdots, B_n$ are activated when they receive $\theta_k$. We specify the exchange of $\theta_k$ by multi-rendezvous among the SBEs. Consequently, $B_{act}$ is represented by the parallel composition of SBEs as follows:

hide $\theta_k$ in $a_1; \cdots; a_i; \theta_k; exit [\{\theta_k\} \cup G_1]$ $\theta_k$; $B_1$ $\mid \{\theta_k\} \cup G_1] \theta_k$; $B_2$ $\mid \{\theta_k\} \cup G_2] \mid \cdots \mid \{\{\theta_k\} \cup G_{n-1}\} \theta_k$; $B_n$).

$TransChoice$: transformation of choice expression

For $TransChoice$, $B_{cho}$ is given as $\sum_{j=1}^{n} B_j$. For each $j$, $B_j$ can be represented by $\prod_{i=1}^{m_j} B'_{j,i}$ where each $B'_{j,i}$ is an SBE. If $m_j = 1$ for each $i$, $B_{cho}$ is an SBE. Here, we consider the case $m_j > 1$ for some $j$ (i.e. parallel operators are specified in $B_j$). We suppose each $B'_{j,i}$ as an action prefixed sequence without losing generality. Let $B_{j,i} \defeq \prod_{i=1}^{m_j} a_{j,i}; B_{j,i}$. We call each $B_j$ as $j$-th group. Let $mx$ be the maximum number among $m_1, \cdots, m_n$.

In choice, any pair from different groups cannot be executed at the same time. Therefore, we can assign $B_{cho}$ to $mx$ SBEs. Although there are various ways of assignment, here we extract an SBE from each group $j (1 \leq j \leq n)$, and compose a new SBE of the choice among the extracted SBEs. For the sake of simplicity, we assign $i$-th elements of all groups to each new $i$-th SBE (if there is no $i$-th element in $j$-th group, exit is used instead). In Fig. 6, the pairs from $B_1 \mid B_2$: $(a_{11}; B_{11}; a_{21}; B_{21})$, $(a_{12}; B_{12}; a_{22}; B_{22})$ and $(exit, a_{23}; B_{23})$ are assigned to new SBEs, where $B_1 := a_{11}; B_{11}$ $\mid a_{12}; B_{12}$ and $B_2 := a_{21}; B_{21} \mid a_{22}; B_{22} \mid a_{23}; B_{23}$.

Next, we introduce a mechanism to keep the equivalence between the new behavior expression and old one. Let us suppose that $j$-th element has executed in one of the new $mx$ SBEs. This means that $j$-th group has selected in $B_{cho}$. So, we have to make only $j$-th element to be executed in all the SBEs after that. To do so, we add extra events to each new SBE for detecting what event has been executed in other SBEs. We achieve those detection by the communication with multi-rendezvous among the new SBEs so that only first executable events in $St(B_{cho})$ synchronize among the SBEs. Accordingly, the following expression is produced for each SBE:

\[
SBE_i \defeq \sum_{j=1}^{n} a_{j,i}; B_{j,i} \mid \sum_{a \in St(B_{j,i}) \; a \neq a_{j,i}} a; a_{j,i}; B_{j,i}
\]

$TransDis$, $TransSeq$: transformation for other operators

In disabling expression $B_1 \trianglerightl B_2$, there is no possibility for SBEs in $B_1$ and those in $B_2$ to be executed simultaneously. So, we assign each pair of the $j$-th elements in $B_1$ and $B_2$ (denoted by $s(b_{1,j}, b_{2,j})$) to a new SBE. In each new SBE, we combine $s(b_{1,j})$ and $s(b_{2,j})$ with choice operators so that the events in $s(b_{1,j})$ should be disabled if an event in $s(b_{2,j})$ is executed. Similarly to the case of choice expressions, each
SBE must be able to detect if such a disabling event is executed in other SBEs. To do so, we add extra events to each SBE for detecting the disabling events in $\text{St}(B_2)$, and specify multi-rendezvous so that those events should be synchronized among the new SBEs. By the above assignment each disabling expression is converted to the parallel composition of $\text{max}(\#\text{Par}(B_1), \#\text{Par}(B_2))$ SBEs.

For sequential expression $B_1 >> B_2$, similarly we extract each pair of SBEs from $B_1$ and $B_2$ and compose a new SBE. Here, we introduce the internal signal $\delta$ to indicate that all events in $B_1$ has been executed and to activate the behavior corresponding to $B_2$. With multi-rendezvous of $\delta$, we make the new SBEs finish the execution for $B_1$ at the same time with starting the execution for $B_2$: $B_1 >> \cdots >> B_1$ can be transformed in the same way.

The detailed transformation algorithms for disabling and sequential expressions are described in Ref. [10] (there, we also give the simplified proof for the correctness of our transformation algorithm).

Example of conversion to synchronous EFSMs

After the algorithm $\text{Trans}$ is applied to the main behavior expression of Fig. 2, $M + 2$ SBEs (corresponding to $\text{Upper}$, $\text{Lower}$ and $M$ instances of $\text{AsgnTek}$) are derived. For example, $\text{Upper}$ is transformed to the following SBE.

$$\begin{align*}
\text{SBE}_{\text{Upper}} &:= \text{label}(\text{Upper}) : a'x : data; g'x'? \text{sum} : \text{bit}; \\
&\quad \text{goto}(\text{Upper})
\end{align*}$$

Here, the recursive process instantiation has been transformed to iteration by label and goto.

3.3 Conversion of each SBE to an EFSM

Each maximal SBE can be converted to an EFSM by the following steps [2]: (1) to replace the variables, events and guard expressions with 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 $s_0$ and the state after the corresponding transition is executed, respectively. Each $\text{goto}(P[G], X:=V)$ is converted to the transition to the state where $\text{label}(P[G])$: $B_P$ is assigned. If the process instantiation changes the values of its process parameters (e.g. $P[G][X] := \overline{B} >> P[G](V)$, the substitutions $X:=V$ are attached to the transition.

3.4 Calculation of multi-rendezvous table

From the syntax tree of the operators among EFSMs and gate names used in each EFSM, we can get the combination of synchronizing EFSMs on each gate. If a multi-rendezvous is specified among a subset of EFSMs $\mathcal{E}$ on gate $g$, we denote that by $\text{Rend}(\mathcal{E}, g)$. The outline for obtaining the multi-rendezvous table is as follows (see [10] for the details): For each $\text{Rend}(\mathcal{E}, g)$, (1) extract events on $g$ from all EFSMs in $\mathcal{E}$; (2) calculate the set of output values $OV$ from the extracted events; (3) compose a rendezvous indication for each value in $OV$ by matching each extracted event with synchronization condition in Table 1.

Some rendezvous indications may contain impossible synchronization tuples, because we don’t use any reachability analysis technique. In this sense, the multi-rendezvous table gives a sufficient condition for multi-rendezvous. However, in the implementation, only the reachable synchronization tuples can be executed as long as each EFSM offers events based on its current state.

4. Merits to use synchronous EFSMs for implementation

In this section, we describe the advantages of using our synchronous EFSMs for the implementation compared with the straightforward implementation of the original LOTOS specification.

4.1 Evaluation on software implementation

We have compared the following two different approaches in terms of execution efficiency: (1) general implementation like in [9] which uses hierarchical data structure corresponding to LOTOS operators for the control among processes; (2) synchronous EFSMs based implementation proposed in this paper. We have experimented the file transfer using the LOTOS specification for the Abracadabra Protocol [5].

The protocol enables a node to successively transfer several files to another node where each file is divided into multiple data packets which are transmitted to lower layer by multi-rendezvous. It is specified in the original specification that eight processes are dynamically created and terminated during each file transfer.

In order to estimate the overhead of dynamic process instantiations/terminations, we have transferred different sizes of files for different times in the experiment as follows: (1) 10 transfer of 100 Kbyte file; (2) 20 transfer of 50 Kbyte file; (3) 50 transfer of 20 Kbyte file; (4) 100 transfer of 10 Kbyte file.

We have generated the object codes from both original specifications and the corresponding synchronous EFSM specifications (in state-oriented style) with our LOTOS compiler [9]. Table 4 shows the CPU time for each case from (1) to (4) with both object codes. According to the table, when the frequency of process instantiation/termination is low (e.g. (1) and (2)), the execution time is similar for both cases. When the frequency is high (e.g. (3) and (4)), the efficiency of synchronous EFSMs based object codes is 20 to 30% better than the others.

4.2 Hardware implementation

The synchronous EFSMs model is also suitable to implement a LOTOS specification as a hardware circuit compared

\[\text{1 we have used 1024 byte packets in the experiment.}\]
In this paper, an algorithm to convert a LOTOS specification to the corresponding synchronous EFSMs has been presented. This conversion is important for a bridge between easiness for writing in a LOTOS specification and easiness for implementation with the Synchronous EFSMs model.

To extend the model and algorithm for a time enhancement of LOTOS will be one of the future work.

References


Fig. 7 A hardware implementation model for synchronous EFSMs.

with its original LOTOS specification.

We have shown the applicability of our model for hardware rapid prototyping of communication protocols in [6]. A hardware implementation of multi-rendezvous for synchronous EFSMs is shown in Fig. 7. It consists of three parts: (i) checking executability for each rendezvous, (ii) avoiding conflict if mutual exclusive multiple rendezvous are executable at the same time, (iii) transferring data for assignments or decision of conditional expressions. For each clock cycle, which multi-rendezvous are executed at that time is decided in those parts. The basic components for each part can be very simple so that these parts are fairly small and fast, even though the number of multi-rendezvous is larger than the original LOTOS specification because all interaction among EFSMs are represented as multi-rendezvous. In the case of a file transfer protocol shown in [6], the size of the circuit for selecting one of appropriate multi-rendezvous is about 300 gates. The size of the whole circuit is about 5000 gates, so the circuit for multi-rendezvous shares few space.

5. Conclusion

In this paper, an algorithm to convert a LOTOS specification to the corresponding synchronous EFSMs has been presented. This conversion is important for a bridge between easiness for writing in a LOTOS specification and easiness for implementation with the Synchronous EFSMs model.

To extend the model and algorithm for a time enhancement of LOTOS will be one of the future work.

References


<table>
<thead>
<tr>
<th>Table 4</th>
<th>CPU time in file transfer (second).</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>original</td>
<td>3.640</td>
</tr>
<tr>
<td>sync. EFSM</td>
<td>3.740</td>
</tr>
</tbody>
</table>


Akira KITAJIMA received the M.E. and Ph.D. degrees in Information and Computer Sciences from Osaka University in 1995 and 1998, respectively. He is now a research assistant of Department of Informatics and Mathematical Science at 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 Informatics and Mathematical Science 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 Informatics and Mathematical Science at 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.