HARDWARE SYNTHESIS FROM PROTOCOL SPECIFICATIONS IN LOTOS

Keiichi Yasumoto[1], Akira Kitajima[2],
Teruo Higashino[2], and Kenichi Taniguchi[2]

[1]Dept. of Information Processing and Management, Shiga Univ.,
Hikone, Shiga 522-0069, JAPAN
yasumoto@biwako.shiga-u.ac.jp

[2]Dept. of Information and Computer Sciences, Osaka Univ.,
Toyonaka, Osaka 560-8531, JAPAN
(kitajima,higashino,taniguchi)@ics.es.osaka-u.ac.jp

Abstract:

In this paper, we propose a technique for hardware implementation of protocol specifications in LOTOS. For the purpose, we define a new model called *synchronous EFSMs* consisting of concurrent EFSMs and a finite set of multi-rendezvous indications among their subsets, and propose a conversion algorithm from a subset of LOTOS. The derived synchronous EFSMs can be easily implemented as a synchronous sequential circuit where all the modules corresponding to the EFSMs work synchronously with the same clock. By applying our technique to the Abracadabra protocol, it is confirmed that the derived circuit handles multi-rendezvous efficiently.

1 INTRODUCTION

Due to the growth of computer networks, efficient implementation of communication protocols has been needed. Thus, the techniques for implementing protocols as hardware circuits have been stressed in recent years.

To specify hardware circuits formally, the description techniques LOTOS [1, 12], Estelle [15] and SDL [13] have been proposed. With these techniques, we can easily describe schemes for hardware circuits using predefined component libraries, and can verify/validate them. However for rapid prototyping, synthesis techniques from the specifications are desirable. Several ideas for hardware synthesis from formal specifications have been proposed [6]. For example, [15] has proposed a synthesis technique from Estelle. However, the technique does not deal with the highly structured specifications containing synchronization among concurrent modules like *multi-rendezvous*. In [7], although a technique to convert timed LOTOS specifica-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35394-4_29

S. Budkowski et al. (eds.), *Formal Description Techniques and Protocol Specification, Testing and Verification*

tions to VHDL specifications has been proposed, only two-way rendezvous between two processes is implemented. [11] has also proposed a technique to synthesize hardware circuits from LOTOS specifications, but focuses only on Basic LOTOS. Since LOTOS specifications are structurally composed of multiple sub-processes which are dynamically invoked, participants of each multi-rendezvous may be decided dynamically each time. In general, for efficient implementation it is desirable that we can transform complicated LOTOS specifications into ones on a flattened model like an EFSM which has no child-parent relationships between processes. It is also important to calculate in advance all the information about the combinations of synchronizing processes, the tuples of synchronizing events and their execution conditions. For the purpose, [10] 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, which needs time proportional to the product of the numbers of events in those processes.

In this paper, we propose a technique for hardware implementation of protocol specifications in a subclass of LOTOS where choice, synchronous/asynchronous parallel, interruption, sequential composition and dynamic process instantiation are specified with data. For the purpose, first we propose a new model called *synchronous EFSMs* [16] for representing LOTOS specifications. Synchronous EFSMs consist of concurrent EFSMs and a finite set of multi-rendezvous indications for their subsets (we call the set a *multi-rendezvous table*). In general, if we use all possible rendezvous instances (i.e., tuples of synchronizing transitions) as the multi-rendezvous table, the number of elements will be quite large. In our model, we reduce the number by composing each rendezvous indication of the tuple of possible event sets. Next, we give an algorithm to derive synchronous EFSMs from a protocol specification in our subclass of LOTOS. In the algorithm, we transform a given LOTOS specification to the parallel composition of EFSMs by introducing internal signals and by replacing each process instantiation with the corresponding behavior expression. We get a multi-rendezvous table statically from the information about transitions in each EFSM, and parallel operators specified among EFSMs by calculating all combinations of EFSMs synchronizing at each gate and by extracting all possible tuples of transitions.

To implement synchronous EFSMs as a synchronous sequential circuit, we compose a module to evaluate whether each rendezvous indication has an executable transition tuple or not. If several mutually exclusive multi-rendezvous become executable simultaneously for some combinations of EFSMs, we select one of them according to a priority order given in advance.

In Sect. 2, we introduce system design in LOTOS and give the definition of synchronous EFSMs. An algorithm to derive synchronous EFSMs is given in Sect. 3. Sect. 4 and 5 present a hardware implementation technique and its evaluation.

2 LOTOS AND SYNCHRONOUS EFSMS

2.1 System design in LOTOS

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

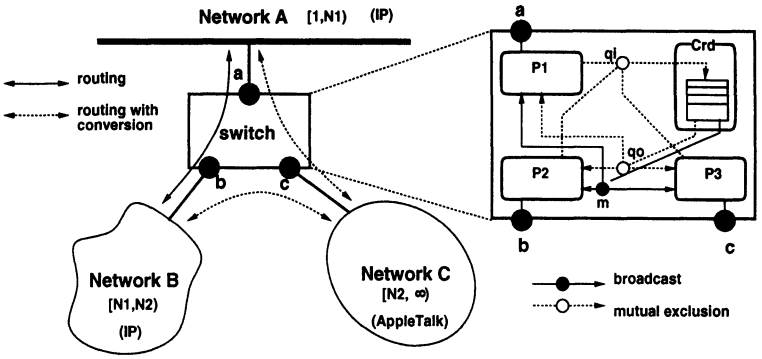


Figure 1 Network switch

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. In a LOTOS specification, we can replace any sub-behavior expression by a process instantiation. It means that we can compose the whole behavior expression as the set of structured modules. (See [3] for details of LOTOS)

Especially, parallel operators and process instantiations make it easy to describe system specifications both structurally and simply. Multi-rendezvous enables group communication among any subset of concurrent processes, which drastically reduces communication actions in specifications. Multi-rendezvous also enables system specifications in constraint/resource oriented style [14] where we can design a system as a set of simple modules, and develop each module independently of the others.

For example, let us design a network switch in LOTOS which has the following requirements: the switch has three ports a, b and c which are connected to different networks A, B and C, respectively (see Fig. 1). Each data packet arriving at the switch should be forwarded to the appropriate network based on its destination. Here, we route the packets based on the intervals: i.e. if the packet's destination is in the interval $[1, N1)$, the packet should be directed towards network A (via port a); if in the interval $[N1, N2)$, towards network B; if in $[N2, \infty)$, towards network C. If the switch receives the packet with the destination 0 (i.e. broadcast), the packet should be broadcasted to all ports except for its reception port. In addition, we suppose networks A and B use the same protocol (e.g. IP), but network C uses a different protocol (e.g. AppleTalk). That means the switch should have the facility for protocol conversion of each packet from either A or B to C (and vice versa).

It is desirable to design the behavior of each port independently of the others. For a better response time, input and output behaviors for each port should be able to work in parallel. In addition, in order to allow asynchrony among ports we use a FIFO queue shared among them. Here, we introduce three internal ports qi, qo and m for the access to the queue. We suppose a new packet can be added to the queue via qi and the packet in the queue can be taken via qo in FIFO manner (see Fig.1). Although m is used in the same way as qo, it is dedicated to the purpose of broadcast.

According to the above discussion, LOTOS processes (`P1` and `P2`) for ports a and b are described as follows:

```
P1[a,qi,qo,m](l1,l2):=I1[a,qi]|||O1[a,qo,m](l1,l2)
where
  I1[a,qi]:= a?idt:IP;qi!idt;I1[a,qi]
  O1[a,qo,m](l1,l2):=qo?odt:IP[l1<=dst(odt)<l2];a!odt;O1[a,qo,m](l1,l2)
       [] m?odt:IP[l1<=src(odt)<l2];O1[a,qo,m](l1,l2)
       [] m?odt:IP[not(l1<=src(odt)<l2)];a!odt;O1[a,qo,m](l1,l2)
P2[b,qi,qo,m](l1,l2):=P1[b,qi,qo,m](l1,l2)
```

(here, $dst(data)$ and $src(data)$ are the ADT functions to get the destination and the source of the packet $data$, respectively. `l1 <= x < l2` represents that the value of x is in the range of $[l1, l2)$. `P2` is described as an instantiation of `P1`)

Inputs/outputs via port c needs protocol conversion. So, for port c, we slightly modify the above process as follows:

```
P3[c,qi,qo,m](l1,l2):=I3[c,qi]|||O3[c,qo,m](l1,l2)
where
  I3[c,qi]:=c?idt:Atk;qi!convIP(idt);I3[c,qi]
  O3[c,qo,m](l1,l2):=
    qo?odt:IP[l1<=dst(odt)<l2];c!convAtk(odt);O3[c,qo,m](l1,l2)
 [] m?odt:IP[l1<=src(odt)<l2];O3[c,qo,m](l1,l2)
 [] m?odt:IP[not(l1<=src(odt)<l2)];c!convAtk(odt);O3[c,qo,m](l1,l2)
```

(here, $convIP()$ means the conversion of the packet to IP, while $convAtk()$ to AppleTalk)

Next, we need a coordinator for the FIFO queue. We describe the queue and its operations by an ADT in LOTOS. The coordinator stores a new packet coming to qi to the queue, or outputs to port either qo or m the last entry in the queue based on its destination. The process Crd for the coordinator can be described as follows:

```
Crd[qi,qo,m](queue):=
    [size(queue)<MAX]->qi?data:IP;Crd[qi,qo,m](append(queue,data))
[] [dst(head(queue))==0]->m!head(queue);Crd[qi,qo,m](tail(queue))
[] [not(dst(head(queue))==0]->qo!head(queue);Crd[qi,qo,m](tail(queue))
```

(here, $size()$, $append()$, $head()$ and $tail()$ represent ADT functions for the queue operation. MAX is the maximum number of entries in the queue)

Finally, we specify the interaction among the above processes. In general, we have to design the mutual exclusion mechanism for the queue since some parallel processes may access it at the same time. However, in LOTOS, we can simply describe such a mechanism with multi-rendezvous as follows (here,`INF` denotes ∞):

```
specification Switch[a,b,c]:=
hide qi,qo,m in
 (P1[a,qi,qo,m](1,N1)|[m]|P2[b,qi,qo,m](N1,N2)|[m]|P3[c,qi,qo,m](N2,INF))
|[qi,qo,m]|
  Crd[qi,qo,m]
```

In the above specification, one of `P1`, `P2` or `P3` synchronizes with `Crd` to store/get a packet to/from the queue via internal ports qi/qo. When the packet's destination is 0, all of `P1`, `P2` and `P3` get the packet at the same time by multi-rendezvous on m.

2.2 Synchronous EFSMs

Synchronous EFSMs are the model where any subset of concurrent EFSMs can communicate with each other via gates by multi-rendezvous[16].

Synchronous EFSMs are given as a set of EFSMs $\{efsm_1, ..., efsm_n\}$ and a multi-rendezvous table $\mathcal{R}$. We suppose that each EFSM can have a finite number of registers,

Table 1 Synchronization condition.

P_i	P_j	condition	result
$g!E_i$	$g!E_j$	$val(E_i) = val(E_j)$	—
$g!E_i$	$g?x : t$	$val(E_i) \in domain(t)$	$x \leftarrow val(E_i)$
$g?x : t$	$g?y : u$	$t = u$	$x, y \leftarrow Ext, Ext \in domain(t)$

($val(E)$ is the normal form of E, Ext is the value input at gate g)

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 for the registers in parallel.

In LOTOS, multi-rendezvous is specified by just giving abstract relationships among concurrent processes by parallel and other operators. For the efficient implementation of multi-rendezvous, we should calculate in advance the information about the combinations of synchronizing EFSMs, the tuples of synchronizing transitions (*synchronization tuples*) and their execution conditions. If we represent the multi-rendezvous simply by the set of all the combinations of transitions in synchronizing EFSMs, the number will be $O(k^n)$ where n and k are the number of EFSMs and the number of transitions in an EFSM, respectively.

Therefore, in our model, we represent all possible multi-rendezvous instances by a set of *rendezvous indications* where each indication is a tuple of transition sets on a gate for a combination of synchronizing EFSMs. Here, every combination of transitions (called *synchronous tuple*) in the sets has the possibility to be executed by multi-rendezvous (i.e. every synchronous tuple satisfies the condition in Table 1).

We denote each rendezvous indication by $\langle (E_1, \cdots, E_m), (A_1, \cdots, A_m) \rangle$ where $(E_1, \cdots, E_m)$ is a tuple of synchronizing EFSMs, and each A_i is the *synchronous transition set* which contains transitions executed in E_i for the rendezvous. We represent elements of A_i as the triples (a, p, I). Here, a 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 undefined variables.

Criteria for each rendezvous indication

To implement multi-rendezvous efficiently, we adopt the following criteria for each rendezvous indication $\langle (E_1, ..., E_m), (A_1, ..., A_m) \rangle$: (1) all transitions in each A_i must be either all input transitions ($g?$) or all output transitions ($g!$); (2) each A_i must not contain transitions with different kind of output values which are executed at the same state (i.e. if two transitions $a!V_1$ and $a!V_2$ are executable at a state, we assign them to different rendezvous indications); and (3) at most one A_i of $(A_1, ..., A_m)$ can have a set of output transitions.

By the above criteria, we can easily know in each rendezvous indication what EFSM outputs some values and what other EFSMs expect the values as their inputs. This contributes to implement data paths to transfer the values among related EFSMs to evaluate the synchronization condition.

The above criteria are not restrictions since we can automatically get the rendezvous indications satisfying them as we will explain in Sect. 3.

By using the above technique, each rendezvous indication $\langle (E_1, \cdots, E_m), (A_1, \cdots, A_m) \rangle$ can represent a maximum of $\prod_{i=1}^{m} |A_i|$ rendezvous instances (here, $|A_i|$ means the number of elements in A_i). Consequently, the number of elements in

the multi-rendezvous table and the time to calculate the table are bound to $O(p \cdot k \cdot n)$ where p is the maximum number of combinations of the synchronizing EFSMs on a gate (usually p can be considered as a constant) and $k \cdot n$ is the sum of the numbers of the output transitions with different values in EFSMs (in the worst case, $k \cdot n$ may be the number of all transitions in EFSMs). For example, for EFSM1$|[a, b]|$EFSM2 in Fig. 4, ten rendezvous instances could be produced. With the above technique, we can reduce the number of tuples to three as shown in Table 3.

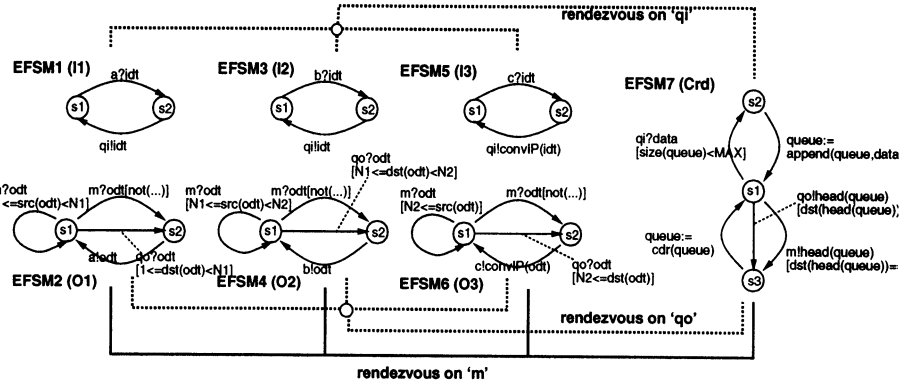
Behavior of synchronous EFSMs

We call each transition e in an EFSM E_i an *asynchronous transition* if $e \notin A_i$ for every rendezvous indication $\langle (E_1, ..., E_m), (A_1, ..., A_m) \rangle \in \mathcal{R}$. We define an asynchronous transition to be *executable* when the current state has the corresponding outgoing edge.

For a rendezvous indication $r = \langle (E_1, \cdots, E_m), (A_1, \cdots, A_m) \rangle$, if each E_i is in a state to execute the transition $e_i \in A_i$ and the execution condition of e_i is *true*, then the synchronous tuple $(e_1, \cdots, e_m)$ can be executed. Whether a synchronous tuple $(e_1, \cdots, e_m)$ can be executed or not is decided by checking the existence of the rendezvous indication which contains such a synchronous tuple. For different two rendezvous indications $r_1 = \langle (E_1, \cdots, E_m), (A_1, \cdots, A_m) \rangle$ and $r_2 = \langle (E'_1, \cdots, E'_l), (A'_1, \cdots, A'_l) \rangle \in \mathcal{R}$, suppose that at least one EFSM is the common member of both r_1 and r_2. In that case, such an EFSM has to decide to execute the transition in either r_1 or r_2. We say that r_1 *conflicts* with r_2 if they can offer the different synchronous tuples at the same time.

Here, we explain how synchronous EFSMs work in cooperation, using an example in Fig. 2. In Fig. 2, the dotted line shows that one of EFSM1, EFSM3 and EFSM5 can synchronize with EFSM7 on gate *qi* at the same time (one of EFSM2, EFSM4, EFSM6 on gate *qo*); the solid line shows that EFSM2, EFSM4, EFSM6 and EFSM7 can synchronize with each other simultaneously on gate m. In the initial state (s_1, s_1, s_1) of EFSM1, EFSM5 and EFSM7, they have the outgoing transitions $a?idt, c?idt$ and $qi?data[size\ (queue) < MAX]$, respectively. The first two transitions are asynchronous, so they are executed independently of the other EFSMs when the input data come to the gates. When $a?idt$ is executed in EFSM1, the current state is changed to (s_2, s_1, s_1). In the state, EFSM1 and EFSM7 have the outgoing edges $qi!idt$ and $qi?data[size(queue) < MAX]$, respectively. Since *queue* contains nothing initially, the execution condition $size(queue) < MAX$ holds. Therefore, the tuple $(qi!idt, qi?data[size(queue) < MAX])$ can be executed by the rendezvous indication (1) of Fig. 2. When the tuple is executed, the value of *idt* is assigned to the undefined variable *data*, and the current state is changed to (s_1, s_1, s_2) in EFSM1, EFSM5 and EFSM7.

In some state, there may be several synchronous tuples to be executable simultaneously. For example, in Fig. 2, when the current state is (s_1, s_1, s_1) for EFSM2, EFSM6 and EFSM7, a synchronous tuple $(qo?odt\ [dst(odt) < N1], qo!head(queue)\ [dst(head(queue))! = 0])$ could be executed between EFSM2 and EFSM7 by the rendezvous indication (4) as well as $(qo?odt\ [N2 \leq dst(odt)], qo!head(queue)\ [dst(head(queue))! = 0])$ between EFSM6 and EFSM7 by the rendezvous indication (6). In that case, one of them must be selected by consensus of the related EFSMs.



No.	tuple of EFSMs	tuple of synchronous transition sets
(1)	(1, 7)	$(\{(qi!idt, true, —)\}, \{(qi?data, size(queue) < MAX, data \leftarrow idt)\})$
(2)	(3, 7)	$(\{(qi!idt, true, —)\}, \{(qi?data, size(queue) < MAX, data \leftarrow idt)\})$
(3)	(5, 7)	$(\{(qi!convIP(idt), true, —)\},$ $\{(qi?data, size(queue) < MAX, data \leftarrow convIP(idt))\})$
(4)	(2, 7)	$(\{(qo?odt, dst(odt) < N1, odt \leftarrow head(queue))\},$ $\{(qo!head(queue), dst(head(queue))! = 0, —)\})$
(5)	(4, 7)	$(\{(qo?odt, N1 \leq dst(odt) < N2, odt \leftarrow head(queue))\},$ $\{(qo!head(queue), dst(head(queue))! = 0, —)\})$
(6)	(6, 7)	$(\{(qo?odt, N2 \leq dst(odt), odt \leftarrow head(queue))\},$ $\{(qo!head(queue), dst(head(queue))! = 0, —)\})$
(7)	(2, 4, 6, 7)	$(\{(m?odt, src(odt) < N1, odt \leftarrow head(queue)),$ $(m?odt, not(src(odt) < N1), odt \leftarrow head(queue))\},$ $\{(m?odt, N1 \leq src(odt) \leq N2, odt \leftarrow head(queue)),$ $(m?odt, not(N1 \leq src(odt) \leq N2), odt \leftarrow head(queue))\},$ $\{(m?odt, N2 \leq src(odt), odt \leftarrow head(queue)),$ $(m?odt, not(N2 \leq src(odt)), odt \leftarrow head(queue))\},$ $\{(m!head(queue), dst(head(queue)) == 0, —)\})$

Figure 2 Example of synchronous EFSMs

3 DERIVING SYNCHRONOUS EFSMS

3.1 Preliminaries

In this paper, we consider any LOTOS specifications represented in the class of Table 2 although we impose the following restrictions on process instantiations.

- Recursive processes are allowed when they are tail recursion (e.g. $P := B >> P$).
- Recursive processes which may produce infinite behavior such as $P := (B1 >> P >> B2)[]exit$ or $P := B\ op\ P$ ($op \in \{[], |[G]|, [>\}$), are not allowed. However, if the recursive process call is guarded and the guard expression can be evaluated statically (e.g. $P(x) := B|||([x < 100]- > P(x + 1))$), we treat such a process.
- Mutually recursive processes are allowed as long as process calls are guarded and the guard expressions can be evaluated statically.

Let $\#Par(B)$ be the function that represents how many parallel processes can be activated at the same time in a behavior expression B. We say a behavior expression B is a *sequential behavior expression (SBE)* if $\#Par(B) = 1$ and B includes

Table 2 Target class of LOTOS

$B0$ ::=	hide G in $B1$ | let *value declarations* in $B1$ | $B1$
$B1$ ::=	$B1 >> B2$ | $B2$
$B2$ ::=	$B2[> B3$ | $B3$
$B3$ ::=	$B3\|\|\|B4$ | $B3\|\|B4$ | $B3\|[G]\|B4$ | $B4$
$B4$ ::=	$B4[]B5$ | $B5$
$B5$ ::=	$[$*boolean exp*$]- > B6$ | $B6$
$B6$ ::=	*event*; B | stop | exit | $(B1)$ | $P[G](el)$
event ::=	*gate* | *gate val* | *gate val* [*boolean exp*]
G ::=	*gate* | *gate*, G
val ::=	?*varname* : *sort val* | !*exp val*
exp ::=	(* every expression written in ACT ONE *)
el ::=	*exp*, *el*

only action prefixed sequences, their choices and iteration. We use $St(B)$ to refer the set of the initially executable events in B. We denote the behavior expression $B_1|[G_1]|...|[G_{m-1}]|B_m$ by $\prod_{i=1}^{m} B_i$, and $B_1[]...[]B_l$ by $\sum_{i=1}^{l} B_i$.

3.2 Transformation algorithm

We transform B_{main} to the parallel composition among SBEs by applying the following operations recursively to its sub-behavior expressions: (1) replace each process instantiation with its behavior expression unless the instantiation appears as a tail recursion (i.e. $P := B >> P$); (2) transform each action prefixed sequence B_{act} such that $\#Par(B_{act}) > 1$ into the parallel composition among sequential behavior expressions (SBEs); (3) transform 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 $P[G](V)$ appearing in its behavior expression B_P as a tail recursion, with the iteration of B_P by introducing **label(P[G]):** and **goto(P[G],X:=V)**. Although other process instantiations are 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 $Trans$ below. Here, *pset* is the set of process instantiations already replaced with their behavior expressions.

```
Algorithm Trans(B, pset)
begin
  if (B = a_1;...;a_l;B', B' ≠ a';B'') then
    if (#Par(B') ≥ 2) then
      Trans(B', pset)
      TransAct(B)
    endif
  else if (B = [guard]- > B') then
    if (guard is true or cannot be calculated statically) then
      Trans(B')
    endif
  else if (B = ∑ B_i) then
    for each i, Trans(B_i, pset)
    TransChoice(B)
  else if (B = B1|[G]|B2) then
    Trans(B1, ∅); Trans(B2, ∅)
```

```
  else if (B = B1[> B2) then
    Trans(B1, pset); Trans(B2, pset); TransDis(B)
  else if (B = B1 >> ... >> Bk) then
    for each i < k, Trans(Bi, ∅)
    Trans(Bk, pset)
    TransSeq(B)
  else if (B = P[G](V)) then
    if (P[G] ∈ 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 ∪ {P[G]})
    endif
  endif
end
```

The sub-procedures used above are given below.

TransAct: **transformation of action prefixed sequence.** For $TransAct$, $a_1; ...; a_l; (B_1|[G_1]|...|[G_{n-1}]|B_n)(\stackrel{def}{=} B_{act})$ 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}, each B_i should be activated after a_l is executed. Accordingly, we assign the event sequence $a_1; ...; a_l$ and the behavior expressions $B_1, ..., B_n$ to different SBEs, respectively. Then, we introduce an internal signal θ so that the SBE for $a_1; ...; a_l$ sends the signal θ to other SBEs after the execution of a_l and that the SBEs for $B_1, ..., B_n$ are activated when they receive θ. We specify the exchange of θ by multi-rendezvous among the SBEs. Consequently, B_{act} is represented by the parallel composition of SBEs as follows:

$a_1; ...; a_l; \theta;$ *exit* $|[\theta]|$ $(\theta; B_1 \,|[\{\theta\} \cup G_1]|\, \theta; B_2 \,|[\{\theta\} \cup G_2]|\, ... \,|[\{\theta\} \cup G_{n-1}]|\, \theta; B_n)$

TransChoice: **transformation of choice expression.** For $TransChoice(B_{cho})$, 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 a SBE. Here, we consider the case $m_j > 1$ for some j (i.e. parallel operators are specified in B_j). We suppose each $B'_{j,i}$ to be an action prefixed sequence without losing generality. Let $B_j \stackrel{def}{=} \prod_{i=1}^{m_j} a_{j,i}; B_{j,i}$. We call each B_j the j-th group. Let mx be the maximum number among $m_1, ..., m_n$.

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 n SBEs. For the sake of simplicity, we assign, to each new i-th SBE, i-th elements of all groups (if there is no i-th element, *exit* is used instead). In Fig. 3, the pairs from $B_1 \,[]\, B_2$: $(a_{11}; B_{11},\ a_{21}; B_{21})$, $(a_{12}; B_{12},\ a_{22}; B_{22})$ and (exit, $a_{23}; B_{23}$) are assigned to new SBEs, where $B1 := a_{11}; B_{11} \,|||\, a_{12}; B_{12}$ and $B2 := a_{21}; B_{21} \,|||\, a_{22}; B_{22} \,|||\, a_{23}; B_{23}$.

Next, we introduce a mechanism to keep the equivalence between the new behavior expression and the old one. Let us suppose that the j-th element has executed in one of the new mx SBEs. This means that the j-th group has selected in B_{cho}. So, we have

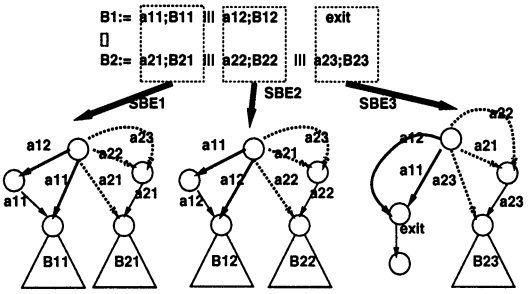


Figure 3 Assignment and resulting SBEs in a choice expression

to make only the j-th element execute in all the SBEs after that. To do so we add, to each new SBE, extra events for detecting what event has executed in other SBEs. We achieve those detections 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:

$$sbe_i \stackrel{def}{=} \sum_{j=1}^{n} a_{j,i}; B_{j,i} \; [] \sum_{a \in St(B_j), a \neq a_{j,i}} a; a_{j,i}; B_{j,i}$$

Fig. 3 shows the resulting SBEs from the above example. Here, a_{11} or a_{12} must be synchronized among the SBEs to detect that the group of $B1$ has been selected (thick solid transitions), while a_{21}, a_{22} or a_{23} detects the group of $B2$ (dotted transitions).

$TransDis, TransSeq$: **transformation for other operators.** In disabling expression $B_1[> B_2$ (here, $B_j \stackrel{def}{=} \prod_{j=1}^{m_j} sbe_{j,i}$), there is no possibility for SBEs in B_1 and those in B_2 to be executed simultaneously. So, we assign to a new SBE each pair of the i-th elements in B_1 and B_2. In each new SBE, we combine $sbe_{1,i}$ and $sbe_{2,i}$ with choice operators so that events in $sbe_{1,i}$ should be disabled if an event in $sbe_{2,i}$ is executed. Similar 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 to each SBE extra events for detecting the disabling events in $St(B_2)$, 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 $max(\#Par(B_1), \#Par(B_2))$ SBEs.

For the 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 internal signal δ to indicate that all events in B_1 have been executed and to activate the behavior corresponding to B_2. With the multi-rendezvous of δ, we make the new SBEs finish the execution of B_1 at the same time as starting the execution of B_2. $B_1 >> \cdots >> B_l$ can be transformed in the same way.

If the sequential expression has tail recursion in the process instantiation such as $P[G](X) := B >> P[G](V)$ and if $\#Par(B) > 1$, all the SBEs extracted from B have to get to their initial states when **goto(P[G],X:=V)** is executed. In that case, for each SBE, we add the transition to the initial state from the state after executing δ with the corresponding subset of the substitutions of X:=V.

The details of the transformation algorithms and a simplified proof for its correctness are given in [16].

Calculation of Multi-Rendezvous Table

Only synchronization operators are specified among EFSMs after applying the transformation algorithm. In this section, we give a technique to get the multi-rendezvous table from transitions in each EFSM and operators specified among EFSMs. From the syntax tree of the operators among EFSMs and gate names used in each EFSM, we can get the combinations 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 *Rend*($\mathcal{E}$, g). The finite set of all synchronous tuples as well as their execution conditions is statically determined if *Rend*($\mathcal{E}$, g) is given for all combinations of $\mathcal{E}$ and g. Let RI be the union of *Rend*($\mathcal{E}$, g) for all $\mathcal{E}$ and g. We calculate the set as follows.

For each *Rend*($\mathcal{E}$, g) $\in RI, \mathcal{E} = \{E_1, \cdots, E_m\}$: (1) extract all transitions on gate g, for each $E_i \in \mathcal{E}$ (let *sync_ev*(E_i, g) be the extracted transitions for E_i); (2) calculate the set of tuples $\{(e_1, \cdots, e_m) | e_i \in$ *sync_ev*$(E_i, g)\}$ where each tuple can satisfy the synchronization condition in Table 1 (let $Tuples(\mathcal{E}, g)$ denote the set).

Next, we convert the synchronous tuples to rendezvous indications as follows, so that they satisfy the criteria in Sect. 2.2. For each $Tuples(\mathcal{E}, g)$:

(i) calculate the set of output values *OV* where each value is assigned to undefined variables by the synchronization.

(ii) for each $E_i \in \mathcal{E}$ and each $v \in OV$, calculate the set of transitions A_i where each transition satisfies the synchronization conditions in Table 1 with v; and compose a rendezvous indication of the tuple $\langle \mathcal{E}, (A_1, \cdots, A_m) \rangle$.

(iii) If A_i includes both input (g?) and output (g!) transitions, we divide such a rendezvous indication $\langle \mathcal{E}, (A_1, \cdots, A_m) \rangle$ to $\langle \mathcal{E}, (A_1, \cdots, A_i^{in}, \cdots, A_m) \rangle$ and $\langle \mathcal{E}, (A_1, \cdots, A_i^{out}, \cdots, A_m) \rangle$ so that each A_i^{in} (A_i^{out}) includes only input (output) transitions. This operation is repeated until the elements of each $A_i (1 \leq i \leq m)$ are all input transitions or all output transitions.

(iv) If $E_i \in \mathcal{E}$ has two different output transitions (i.e. $g!V_1, g!V_2$) in A_i from the same state, the rendezvous indication $\langle \mathcal{E}, (A_1, \cdots, A_i, \cdots, A_m) \rangle$ is divided into $\langle \mathcal{E}, (A_1, \cdots, A_i - \{g!V_1\}, \cdots, A_m) \rangle$ and $\langle \mathcal{E}, (A_1, \cdots, A_i - \{g!V_2\}, \cdots, A_m) \rangle$. This operation is repeated until each A_i has only one output transition from each state in E_i.

(v) If a rendezvous indication $\langle \mathcal{E}, (A_1, \cdots, A_m) \rangle$ contains A_i and A_j such that the both sets have output transitions, we modify a transition in either A_i or A_j to an input transition. For example, $a!x \; ||[a]|| \; a!y \; ||[a]|| \; a?z$ is transformed to $a!x \; ||[a]|| \; a?w \, [w = y] \; ||[a]|| \; a?z$.

The above procedure could produce some rendezvous indications which contain impossible synchronous tuples since we do not use any reachability analysis technique. A synchronous tuple can be executed as long as the synchronization condition of the tuple holds for a rendezvous indication. Not all synchronous tuples in each rendezvous indication can be executed. That means the multi-rendezvous table itself is a sufficient condition for implying the possible multi-rendezvous instances. In our model, however, only the valid synchronous tuples become executable since the executability of each rendezvous indication is evaluated at each reachable state of EFSMs.

3.3 Example of conversion to synchronous EFSMs

After the algorithm $Trans$ is applied to the main behavior expression of $Switch$ in Sect. 2.1, seven SBEs are derived. We show part of them below:

SBE_{I1} := label(I1): a?idt:IP; qi!idt; goto(I1,—)
SBE_{O1} := label(O1): (qo?odt:IP[1 ≤ dst(odt) < N1]; a!odt; goto(O1,—)
[] m?odt:IP[1 ≤ src(odt) < N1]; goto(O1,—)
[] m?odt:IP[not(1 ≤ src(odt) < N1)]; a!odt; goto(O1,—))
SBE_{Crd} := label(Crd): (qi?data:IP[size(queue)<MAX]; goto(Crd,queue:=append(queue,data))
[] m!head(queue)[dst(head(queue))==0]; goto(Crd,queue:=cdr(queue))
[] qo!head(queue)[dst(head(queue))!=0]; goto(Crd,queue:=cdr(queue)))

In the above SBEs, process instantiations in the form of tail recursions are converted to the appropriate **goto** transitions. Each derived SBE can be converted to an EFSM easily. Fig. 2 depicts the resulting EFSMs and the multi-rendezvous table. Here, EFSM1 – EFSM7 are converted from SBE_{I1} – SBE_{Crd}, respectively.

4 HARDWARE SYNTHESIS FROM SYNCHRONOUS EFSMS

In this section, we give the technique to convert given synchronous EFSMs into a synchronous sequential circuit (our preliminary work can be found in [5]). Hereafter, we suppose the modules corresponding to EFSMs work synchronously with the same clock. In each clock cycle, each EFSM can execute a transition as long as its execution condition holds. We assume the components corresponding to ADT functions (e.g. guard expressions) are provided as combinational logic circuits and they can output the resulting values within a clock cycle. The circuit for each EFSM can be implemented easily by well-known techniques [9]. So, here we concentrate on the implementation of multi-rendezvous among EFSMs.

Given EFSMs and a multi-rendezvous table, we implement multi-rendezvous among EFSMs as the *multi-rendezvous circuit* consisting of the following three sub-parts: *(1) executability check part* checking whether there exist executable synchronous tuples for each rendezvous indication at each state; *(2) data transfer part* transferring the required data from a certain EFSM to the other EFSMs so that each EFSM can calculate the execution condition (guard) of its transition. *(3) conflict avoidance part* selecting a synchronous tuple among some mutually exclusive synchronous tuples;

Hereafter, we suppose that synchronous EFSMs are given as $\langle EFSM, \mathcal{R} \rangle$ where each rendezvous indication $r \in \mathcal{R}$ is represented as $\langle (E_1, ..., E_m), (A_1, ..., A_m) \rangle$ where each $E_i \in EFSM$.

Constructing executability check and data transfer parts

For the executability checking part, every EFSM in each rendezvous indication must check whether some transitions in its synchronous transition set are executable at the current state. So in each E_i, for every $r \in \mathcal{R}$, we provide a circuit generating an output signal r_i_ok which becomes *true* (i.e. 1) only when a transition in A_i becomes executable. Consequently, for the rendezvous indication r there exist some executable synchronous tuples if and only if r_1_ok, ..., r_m_ok (denoted by r_*_ok) are *true*.

For the data transfer part, EFSMs with input transitions and an EFSM with output transitions can be determined statically for each rendezvous indication by the criteria

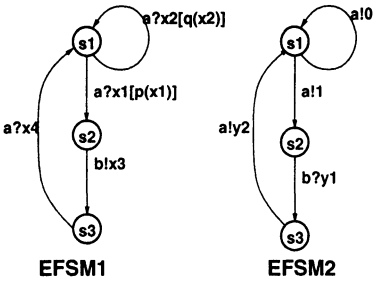


Figure 4 An example of EFSMs.

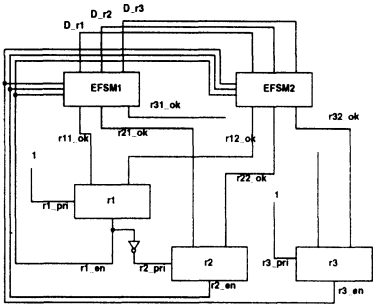


Figure 5 An example of derived circuit.

in Sect. 2.2. Hence, we provide a path D_r among EFSMs for each r so that an EFSM outputs an appropriate value to the path and the others obtain the value.

Constructing conflict avoidance part

The conflict avoidance part generates the signal r_en which becomes *true* when r has the right to execute its synchronous tuple, avoiding conflicts between r and other exclusive rendezvous indications. Although there can be some policies for avoiding conflicts, we adopt a policy that gives a priority (or total order) among rendezvous indications and selects a rendezvous indication by the priority.

Synchronous tuples of rendezvous indication r_1 and r_2 can conflict if and only if there exists the case that for a state set $(s_1, \cdots, s_p)$ of the common EFSMs in both rendezvous indications, the synchronous transition tuples 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 indications, we can statically determine whether they can conflict or not by checking the given EFSMs and the multi-rendezvous table.

Any synchronous tuples of r cannot be executed when another conflicting rendezvous indication with priority higher than r is ready to execute a synchronous tuple. Consequently, we construct r_en as follows:

$r\_en = r_1\_ok \wedge \cdots \wedge r_m\_ok \wedge pri_r$ where $pri_r = \neg r^1\_en \wedge \cdots \wedge \neg r^h\_en$

(Here, $\{r^1, \cdots, r^h\}$ are the rendezvous indications with higher priorities than r which conflict with r. pri_r means whether r has the right to execute its synchronous tuple or not).

Although we have introduced a priority based method, a module generating random numbers can also be used to select one of the conflicting rendezvous.

An example of derived circuit

In this section, we explain how we can derive the circuit in Fig. 5 from the synchronous EFSMs in Fig. 4 and the multi-rendezvous table in Table 3.

Hereafter, we denote the output signal from $EFSM_j$ for the rendezvous indication r_i as r_{ij}_ok. At the initial state (s_1, s_1), EFSM1 first calculates the output value r_{11}_ok for the rendezvous indication r_1 as follows. As EFSM1's current state is s_1, EFSM1 calculates the execution condition $p(x_1) \vee q(x_2)$ for the transitions $a?x_1\,[p(x_1)]$ and $a?x_2\,[q(x_2)]$ which are transitions in its transition set and that of r_1 respectively. Furthermore, since $p(x_1)$ and $q(x_2)$ need external values to calculate the conditions,

Table 3 An example of multi-rendezvous table.

Info. No.	EFSMs	synchronizing transition sets
r_1	(EFSM1, EFSM2)	$(\{(a?x_1, p(x_1), x_1 \leftarrow 1 or g(y)),$ $(a?x_2, q(x_2), x_2 \leftarrow 1$ *or* $g(y)), (a?x_3)\}, \{(a!1, true, -)\})$
r_2	(EFSM1, EFSM2)	$(\{(a?x_1, p(x_1), x_1 \leftarrow 0),$ $(a?x_2, q(x_2), x_2 \leftarrow 0), (a?x_3)\}, \{(a!0, true, -)\})$
r_3	(EFSM1, EFSM2)	$(\{(b!f(x_1), true, -)\}, \{(b?y, true, y \leftarrow f(x_1))\})$

EFSM1 uses the value from the data path D_{r1} (the data path for the rendezvous indication r_1) for the values of x_1 and x_2. So, it outputs the value of $p(D_{r1}) \vee q(D_{r1})$ to r_{11}_ok. On the other hand, as EFSM2's current state is s_1, EFSM2 can execute the transitions a!1 in its transition set for the rendezvous indication r_1. So it outputs *true* to r_{12}_ok. In addition, since $a!1$ is an output transition, it outputs the value 1 to the data path D_{r1}. For other rendezvous indications, EFSMs do the same operation. If r_{11}_ok and r_{21}_ok are both *true*, r_1 and r_2 are conflict with each other because both r_{12}_ok and r_{22}_ok are also *true*. In this case, the conflict avoidance part selects r_1 and outputs *true* only to r_1_en (since we assume that the priority $r_1 > r_2 > r_3$ holds).

Finally, if r_1_en is *true*, EFSM1 executes either $a?x_1$ or $a?x_2$ depending on which condition in $p(x_1)$ and $q(x_2)$ holds (when the both conditions are *true*, EFSM1 selects one of them by itself). If $a?x_2$ is executed, the value from D_{r1} is assigned to x_2.

Further optimization

Clock frequency is an important factor for efficient circuits. To what extent the frequency can go up depends on the *critical path* of the circuit, which is the most time consuming path of the logic gates used in a clock cycle.

In our technique, for each rendezvous indication r_i, (i) r_i_ok and (ii) r_i_en have to be evaluated in a clock cycle. The evaluation for (i) requires the transfer of data values among the related EFSMs and the evaluation of the guard expression in each EFSM. The evaluation for (ii) requires the logic gates of $\log h$ depth where h is the number of conflicting rendezvous indications. To shorten the critical path, we can take the following approach: (1) divide each complicated ADT function into several sub-modules with registers to calculate the result in several clock cycles based on the technique in [2]. (2) solve each conflict among multiple rendezvous indications in several clock cycles, for example, by dividing them into several groups.

Another topic of the optimization is to reduce the number of data paths to simplify the resulting circuits. In general, several rendezvous indications can share a data path as long as they do not conflict with each other or cannot be executed at the same time. To share the data path, we allocate a bus available to the related EFSMs for some rendezvous indications. With this technique, all the required data path can be implemented by just N data buses where N is the maximum number of conflicting/simultaneous rendezvous indications.

5 EXPERIMENTAL RESULTS AND DISCUSSION

We have applied our technique to the LOTOS specification of Abracadabra protocol [4] and constructed the hardware circuit to show that the constructed circuit is reasonably small and fast. We have used a hardware synthesis system PARTHENON [8] which has been developed by NTT.

In general, the modules for ADT functions (e.g. execution conditions) depend on how to implement the data types in the target circuit (e.g. the size of each data). Therefore, in this experiment, we have mainly evaluated the derived multi-rendezvous circuit without the modules for calculating ADT functions in each EFSM.

The greatest effect in the performance will be the maximum depth of the logic gates in the circuit. Eight EFSMs were derived from the LOTOS specification of Abracadabra protocol using the algorithm in Sect. 3 (the derived synchronous EFSMs can be found in [5]). The number of rendezvous indications was 85, and the maximum number of the depth of the logic gates for the multi-rendezvous circuit was 7. The maximum depth became six after some optimization.

Another criterion should be the size of the resulting circuit. The size of the multi-rendezvous circuit grows in proportion to the number of rendezvous indications since each rendezvous indication has its own module. The time for selecting a synchronous tuple set grows in proportion to the maximum number of rendezvous indications which may conflict with each other. In general, the number of the logic gates in the circuit depends on the size of data.

We have synthesized the whole circuit with ADT data/functions using 8 bit data. We have used several hardware modules for implementing ADT functions such as integer comparison (e.g. $=, \leq$) and addition (e.g. $inc, +$).

In that case, the whole circuit obtained has about 5000 gates: about 500 gates for ADT functions; about 300 gates for the multi-rendezvous circuit; and the remainder for registers, selectors and control signals for EFSMs.

The maximum depth of logic gates for ADT functions was 15. In the experiment, our technique requires additional 6 logic gates in depth for the multi-rendezvous circuit. That means the whole circuit could work with the clock frequency at least 70 % as high as in hardware circuits without multi-rendezvous. In fact, as we explained in the previous section, the multi-rendezvous circuit could be optimized further for practical use as well as other modules. We can approximately estimate the optimized performance from the circuit automatically derived with our technique. According to the above discussion, we think our technique can be used for rapid prototyping.

6 CONCLUSION

In this paper, we have proposed a hardware implementation technique from LOTOS specifications. In the technique, by composing each rendezvous indication of the tuple of event sets, we can keep the information about all possible rendezvous instances in a reasonable space. It is important that our conversion algorithm does not require any reachability analysis among parallel processes. Such analysis needs plenty of time proportional to the product of the number of events in parallel processes. Through the experiment for the Abracadabra protocol, we have confirmed our technique can be used for the rapid prototyping. We are going to evaluate our technique by implementing various protocols/systems, and possibly develop the optimization techniques for synthesized circuits for their practical use. To apply the technique to a time extension of LOTOS is part of future work.

References

[1] Faci, M. and Logrippo, L.: "Specifying Hardware Systems in LOTOS", *Proc. IFIP Int. Conf. on Hardware Description Languages and Applications (CHDL'93)*: 305 – 312 (1993).

[2] Higashino, T., Yasumoto, K., Kitamichi, J. and Taniguchi, K. : "Hardware Synthesis from a Restricted Class of LOTOS Expressions", *Proc. 14th IFIP Int. Symp. on Protocol Specification, Testing, and Verification (PSTV-XIV)*: 355–362 (1994).

[3] ISO: "Information Processing System, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", *ISO 8807* (1989).

[4] ISO/IEC/TR 10167: "Information Technology – Open Systems Interconnection – Guidelines for the application of Estelle, LOTOS and SDL" (1991).

[5] Kitajima, A., Yasumoto, K., Higashino, T. and Taniguchi, K.: "A Method to Convert Concurrent EFSMs with Multi-Rendezvous into Synchronous Sequential Circuits", *IEICE Trans. on Fundamentals*, E81-A (4): 566 – 575 (1998).

[6] Kloos, C. D. and Damm, W. (Eds.): "Practical Formal Methods for Hardware Design", *Research Reports Esprit, Project 6128, FORMAT*, Vol. 1, Springer (1996).

[7] Lopez, A. M., Kloos, C. D., Valladares, T. R. and Moro, T. M.: "Generating VHDL code from LOTOS Descriptions", in [6]: 266 – 293 (1996).

[8] Nakamura, Y. : "An Integrated Logic Design Environment Based on Behavioral Description", *IEEE Trans. on CAD*, CAD–6 (3): 322–336 (1987).

[9] Ott, D. E. and Wilderotter, T. J. : "A Designer's Guide to VHDL Synthesis", *Kluwer Academic Publishers* (1994).

[10] Quemada, J., Larrabeiti, D. and Pavon, S. : "Compressing the State Space Representation of LOTOS Specifications", *Proc. 6th IFIP Int. Conf. on Formal Description Techniques (FORTE'93)*: 19–34 (1993).

[11] Sisto, R. : "A method to build symbolic representations of LOTOS specifications", *Proc. 15th IFIP Int. Symp. on Protocol Specification, Testing and Verification (PSTV-XV)*: 331–346 (1995).

[12] Turner, K. J. and Sinnott, R. O.: "DILL: Specifying Digital Logic in LOTOS", *Proc. 6th IFIP Formal Description Techniques (FORTE'93)* (1994).

[13] Csopaki, G. and Turner, K. J.: "Modelling Digital Logic in SDL": *Proc. Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'97)*: 367 – 382 (1997).

[14] Vissers, C. A., Scollo, G. and Sinderen, M. v.: "Architecture and Specification Style in Formal Descriptions of Distributed Systems", *Proc. 8th Int. Symp. on Protocol Specification, Testing, and Verification (PSTV-VIII)*: 189 – 204 (1988).

[15] Wytrebowicz, J. : "Hardware Specification Generated from Estelle", *Proc. 15th IFIP Int. Symp. on Protocol Specification, Testing and Verification (PSTV-XV)*: 435–450 (1996).

[16] Yasumoto, K.: "A method to derive concurrent synchronous EFSMs from protocol specifications in LOTOS", *Tech. Rep. # 1113 of Dept. IRO, University of Montreal* (1998).