理論計算機科学とその周辺
研究集会 1992.1.29-31

LOTOSによる分散システムの全体記述と各ノードの動作記述 - 等価性と変換アルゴリズムについて -

東野輝夫，安本慶一，谷口健一
大阪大学基礎工学部情報工学科

Service Specification and Protocol Specifications in LOTOS - Equivalence and Synthesis -

Teruo Higashino, Keiichi Yasumoto and Kenichi Taniguchi
Department of Information and Computer Sciences
Osaka University, Osaka 560 Japan

Abstract

LOTOS is a language developed within ISO for the formal description of communication protocols and distributed systems. In LOTOS, requirements for a distributed system are called a "service specification". Each node exchanges synchronization messages to ensure the temporal ordering for the execution of events in a service specification. The actions of each node are described as a "protocol specification". In this paper, we introduce a method to derive protocol specifications from a service specification written in a LOTOS based language. In order to derive the protocol specifications, we make the syntax tree of a given service specification and give some attributes for each node in the tree. The protocol specifications are derived automatically by evaluating these attributes. The derived protocol specifications satisfy the given service specification. We also explain a LOTOS simulator for the execution of derived protocol specifications.

1. Introduction

LOTOS(1-4) is a language developed within ISO for the formal description of communication protocols and distributed systems. Recently, the specifications of many OSI protocols(5) are described in LOTOS(2,6). Requirements for a distributed system are described as a "service specification"(7). In LOTOS, service primitives of each node in a distributed system are called "events", and the temporal ordering of the execution of events are described as a service specification. On the protocol level, several nodes cooperate to provide the required service. They exchange synchronization messages to ensure the temporal ordering of the execution of events through a communication medium. In the communication medium, we assume that there is a communication channel from each node "i" to any other node "j", and that the communication channel is modeled as a FIFO queue whose capacity is infinite. The actions of each node are described as a "protocol specification". That is, a protocol specification of a node specifies the temporal ordering of the execution of both the events of the node and sending/receiving interactions of synchronization messages.

In order to get protocol specifications satisfying a given service specification, there are two techniques : (1)analysis and (2)synthesis. Verification and testing are analysis techniques. These techniques are used for detecting design errors such as deadlocks, unspecified receptions and so on. Although some analysis techniques have been proposed to determine whether given protocol specifications satisfy a service specification, usually it takes much time to ensure that given protocol specifications satisfy a service specification. As a technique to design distributed systems, it is desirable that the designer describes only a service specification and protocol specifications can be derived from the service specification automatically. Some synthesis techniques have been proposed(8-14). In this paper, we will introduce a synthesis technique to derive protocol specifications from a service specification written in a LOTOS based language. This synthesis technique uses only service specifications and it does not require any further information. The technique has been proposed in Ref. (15) and extended in Ref. (16-21). This paper gives a survey for this synthesis technique.

In order to observe the execution of LOTOS programs, some LOTOS simulators (interpreters) have been developed(4,22,23). These simulators are used for simulating the execution of service specifications. In order to simulate the execution of protocol specifications, we need the facility for the exchange of synchronization messages in addition to the ordinary facilities of LOTOS simulators. We have developed a LOTOS simulator *PROSPEX* (PROtocol SPecification EXecutor) for the execution of protocol specifications(24). Suppose that there are N nodes in a distributed system. We use N *PROSPEX* to simulate N protocol specifications. That is, each *PROSPEX* simulates a protocol specification and exchanges synchronization messages each other.

In Section 2, we introduce a LOTOS based language for describing service specifications. In Section 3, a method for deriving protocol specifications from a service specification is explained. Our LOTOS simulator *PROSPEX* is introduced in Section 4.

2. Service Specifications

2.1 LOTOS

In LOTOS, a distributed system is described as a collection of processes. A special process is treated as the main process. A process consists of a behavior expression (a sequence of events and operators) where some operators define the temporal ordering of the execution of events. Let P and B be a process and a behavior expression, respectively. A process definition is described as "P := B". If the process P is invoked, then the events in the behavior expression B are executed. The following operators are used in behavior expressions.

(1) The sequential execution of simple interactions ";"

A behavior expression "a ; B" represents that the behavior expression "B" is executable after the event "a" is executed.

(2) Non-deterministic choice of alternatives "[]"

A behavior expression "B_1 [] B_2" represents that only one of the behavior expressions "B_1" and "B_2" is executed. If an event in "B_1" is executed, then only the events in "B_1" are executable and the events in "B_2" are not executed.

(3) Independent parallelism "|||"

A behavior expression "B_1 ||| B_2" represents that both behavior expressions "B_1" and "B_2" are executable in parallel. The events in "B_1" and "B_2" are executed independently.

(4) Dependent parallelism with rendezvous interactions "||"

A behavior expression "B_1 |[g_1,..,g_n]| B_2" represents that both behavior expressions "B_1" and "B_2" are executable in parallel. The events in "B_1" and "B_2" belonging to {g_1,...,g_n} must be executed as rendezvous interactions. If all events in "B_1" and "B_2" are contained in {g_1,...,g_n}, then "B_1 |[g_1,..,g_n]| B_2" may be described as "B_1 || B_2".

(5) Sequential composition ">>"

A behavior expression "B_1 >> B_2" represents that the behavior expression "B_2" is executable after the execution of the behavior expression "B_1" is finished successfully.

(6) Disabling operator "[>"

Disabling operator "[>" represents the interruption of a particular sequence of events by a disabling event. A behavior expression "B_1 [> B_2" represents that the behavior expression "B_1" is executable until an event "d" of the behavior expression "B_2" is executed. If "d" is executed, then only the events in "B_2" are executable.

As another operator, there is the hiding operator (see Ref. (1,4)). An algebraic language ACT ONE(25) is used to represent the values and data structures in LOTOS. A sub-language ignoring the values and data structures in LOTOS is called "Basic LOTOS".

2.2 Language to Describe Service Specifications

The specification language used in our derivation algorithm is functionally equivalent to Basic LOTOS, except that the disabling operator and hiding operator are not supported. The language is used to describe both service specifications and derived protocol specifications.

Suppose that a LOTOS program L consists of the tuple L=<P_1,P_2,...,P_n> of n processes P_1, P_2,...,P_n, and that the first process P_1 is the main process. We define the syntax of each process definition using the production rules (1)~(12) in Table 1. In Table 1, "Process_def" is the starting symbol. "Proc_Id" and "Event_Id" must be defined as identifiers using some terminal symbols. The keywords "process", ":=" and "endproc" and the operators ">>", "||", "|||", "[]", ";" and "exit" are treated as terminal symbols. "Event_subset" is a set of "Event_Id". A "Proc_Id" denotes a process, which is written as an "Identifier". An "Event_Id" may denote either:

- a service primitive interaction : It is written as "$\text{Identifier}^{\text{Node}}$" where "Identifier" denotes the service primitive itself and "Node" denotes the node name at which the interaction takes place. For example, "a^2" denotes the service primitive "a" at the node 2 (here, we assume that each interaction takes place at only one node).
- an interaction of sending message : It is written as "$s_i(m)$" which means the sending of the message "m" to the node "i".
- an interaction of receiving message : It is written as "$r_i(m)$" which means the receiving of the message "m" from the node "i".

Interactions of sending/receiving messages are only used in protocol specifications.

2.3 Example of Service Specifications

Let us consider an example in Fig. 1. In Fig. 1, there are 3 nodes. Suppose that the user wants to copy some elements in a file of the node 1 into another file of the node 3, but the reverse order. At the node 1, we can only execute "read^1" which is a service primitive interaction reading a element from the file. At the node 3, we can only execute "write^3" which is a service primitive interaction writing a receiving element into the file. The node 2 has a stack. At the node 2, we can execute either "push^2" or "pop^2". "Push^2" inserts the last receiving element in the local stack. "Pop^2" extracts the past pushed element from the local stack. For simplicity of the explanation, we do not consider the contents of the elements. A service specification of this example is written as follows using the production rules in Table 1 :

L = <A>
process A := (read^1 ; push^2 ; A >> pop^2 ; write^3 ;exit)
[] read^1 ; write^3 ; exit endproc

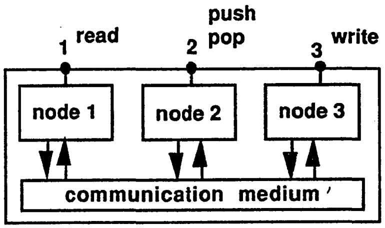


Fig. 1: Example of Service

3. Derivation of Protocol Specifications

In this section, we will explain a method for the derivation of protocol specifications. The derivation algorithm of this paper is the extended version of those in Ref. (15,18). For more complex service specifications containing the disabling operator and data parameters, see the derivation algorithms in Ref. (19,20).

3.1 Example of Protocol Specifications

First, we will give an example of protocol specifications. For the service specification L in Section 2.3, we derive the following protocol specifications L_1, L_2 and L_3. Here, some integers such as 8, 14 and 17 are used as the synchronization messages.

Node 1:
L_1 = <A>
process A := read^1; $s_2(14)$; $r_2(17)$; A
[] read^1; $s_3(35)$; $s_2(8)$; exit endproc

Node 2:
L_2 = <A>
process A := ($r_1(14)$; push^2; ($s_1(17)$; exit ||| $s_3(17)$; exit)
>> A >> $r_3(11)$; pop^2; $s_3(25)$; exit)
[] $r_1(8)$; exit endproc

Node 3:
L_3 = <A>
process A := ($r_2(17)$; A >> $s_2(11)$; $r_2(25)$; write^3; exit)
[] $r_1(35)$; write^3; exit endproc

Fig.2 represents an execution process of these protocol specifications. The dotted lines in Fig. 2 denote the exchange of synchronization messages.

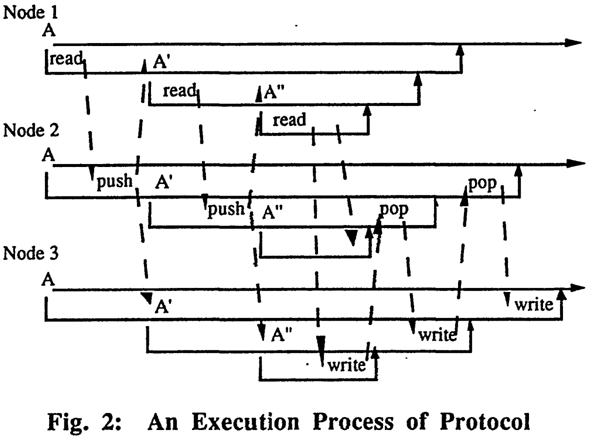


Fig. 2: An Execution Process of Protocol Specifications

Table 1: Syntax of Specification Language and Attribute Evaluation Rules

Nr.	Production Rules	Attribute SP (Starting Places)
(1)	Process_def --> process Proc_Id := e endproc	SP(Process_def) = SP(e)
(2)	e_1 --> Par >> e_2	SP(e_1) = SP(Par)
(3)	e --> Par	SP(e) = SP(Par)
(4)	Par$_1$ --> Choice |[event_subset]| Par$_2$	SP(Par$_1$) = SP(Choice) $\cup$ SP(Par$_2$)
(5)	Par$_1$ --> Choice ||| Par$_2$	SP(Par$_1$) = SP(Choice) $\cup$ SP(Par$_2$)
(6)	Par --> Choice	SP(Par) = SP(Choice)
(7)	Choice$_1$ --> Seq [] Choice$_2$	SP(Choice$_1$) = SP(Seq) = SP(Choice$_2$)
(8)	Choice --> Seq	SP(Choice) = SP(Seq)
(9)	Seq$_1$ --> Event_Id ; Seq$_2$	SP(Seq$_1$) = {place(Event_Id)}
(10)	Seq --> Event_Id ; exit	SP(Seq) = {place(Event_Id)}
(11)	Seq --> Event_Id ; Proc_Id	SP(Seq) = {place(Event_Id)}
(12)	Seq --> (e)	SP(Seq) = SP(e)

	Attributes EP (Ending Places) and AP (All Places)	
	EP	AP
(1)	EP(Process_def) = EP(e)	AP(Process_def) = AP(e)
(2)	EP(e_1) = EP(e_2)	AP(e_1) = AP(Par) $\cup$ AP(e_2)
(3)	EP(e) = EP(Par)	AP(e) = AP(Par)
(4)	EP(Par$_1$) = EP(Choice) $\cup$ EP(Par$_2$)	AP(Par$_1$) = AP(Choice) $\cup$ AP(Par$_2$)
(5)	EP(Par$_1$) = EP(Choice) $\cup$ EP(Par$_2$)	AP(Par$_1$) = AP(Choice) $\cup$ AP(Par$_2$)
(6)	EP(Par) = EP(Choice)	AP(Par) = AP(Choice)
(7)	EP(Choice$_1$) = EP(Seq) = EP(Choice$_2$)	AP(Choice$_1$) = AP(Seq) $\cup$ AP(Choice$_2$)
(8)	EP(Choice) = EP(Seq)	AP(Choice) = AP(Seq)
(9)	EP(Seq$_1$) = EP(Seq$_2$)	AP(Seq$_1$) = {place(Event_Id)} $\cup$ AP(Seq$_2$)
(10)	EP(Seq) = {place(Event_Id)}	AP(Seq) = {place(Event_Id)}
(11)	EP(Seq) = EP(Proc_Id)	AP(Seq) = {place(Event_Id)} $\cup$ AP(Proc_Id)
(12)	EP(Seq) = EP(e)	AP(Seq) = AP(e)

(Here, place(Identifier^{Node}) = Node)

3.2 Principles for Deriving Protocol Specifications

In this section, we will explain the principles for deriving protocol specifications. The basic idea of the derivation is to use the notion of "projection". That is, first, the events of a node "p" are selected from a given service specification, and then the sending/receiving interactions of synchronization messages between the node "p" and other nodes are added.

3.2.1 Attributes

The information concerning the exchange of synchronization messages is implicitly defined in each service specification. This information is found by assigning some attributes to the nonterminal symbols of the syntax tree of the service specification. See Ref. (26) for details of attribute grammars. In this paper, we use the following three attributes (here, exp(x) represents the behavior expression which is derived from the nonterminal symbol "x").

SP(x) : The set of nodes where the first events of exp(x) are executed. It is called *Starting Places* of the nonterminal symbol "x".

EP(x) : The set of nodes where the last events of exp(x) are executed. It is called *Ending Places* of "x".

AP(x) : The set of all nodes where the events of exp(x) are executed. It is called *All Places* of "x".

These attributes are calculated as the synthesized attributes using the attribute evaluation rules in Table 1. The syntax tree for the process A described in Section 2.3 and the attributes for some nonterminal symbols of the tree are described in Fig. 3. Although the parameter "x" of the attributes SP(x), EP(x) and AP(x) is a nonterminal symbol, we may use exp(x) instead of "x". That is, we may use SP(exp(x)) instead of SP(x) if there is no confusion.

We give the attributes SP(x), EP(x) and AP(x) not only for the nonterminal symbols but also for the leaves corresponding to event identifiers and process identifiers. If "Event_Id" is "Identifier^{Node}", then we define SP(Event_Id) = EP(Event_Id) = AP(Event_Id) = {Node}. The attributes corresponding to process identifiers are treated as variables. We equate the variables of such a leaf node, for instance A, with the values obtained by synthesis for the root node "Process_def" corresponding to the same process identifier A. If the equation "SP(A) = SP(A) $\cup$ X" holds, then "SP(A) = X" is obtained as the solution. For the process A in Fig. 3, the attribute SP(A), EP(A) and AP(A) are treated as variables. We find the equations "SP(A) = {1}", "EP(A) = EP(A) $\cup$ {3}" and "AP(A) = AP(A) $\cup$ {1,2,3}". Therefore, the solutions are "SP(A) = {1}", "EP(A) = {3}" and "AP(A) = {1,2,3}". These attributes are used to determine which nodes need to synchronize their events.

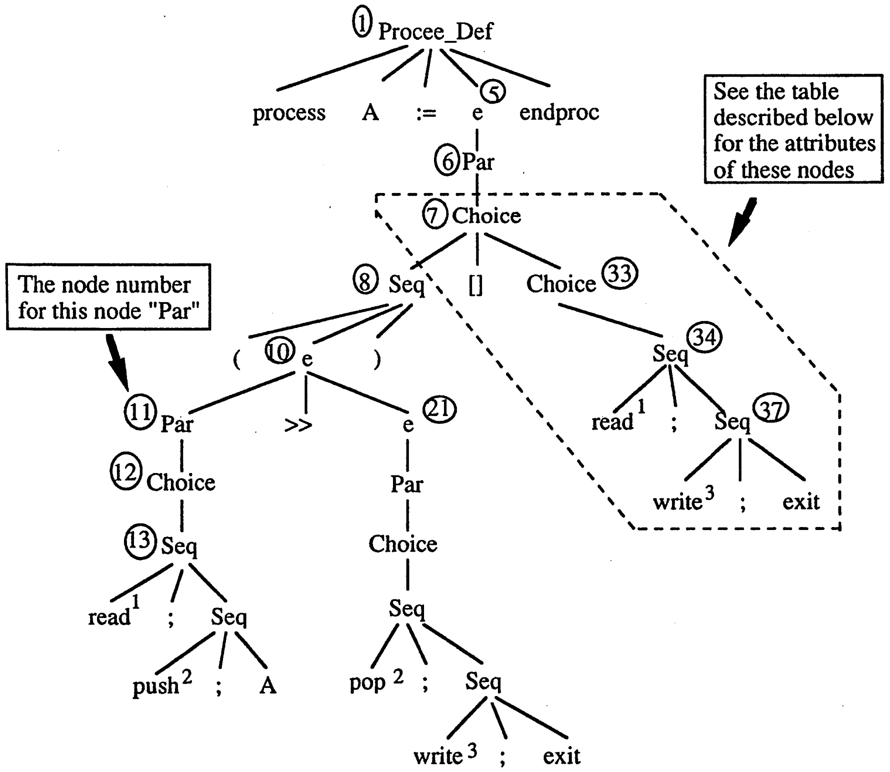
3.2.2 Basic idea of Derivation

(1) The sequential execution ";" and ">>"

For a behavior expression of the form "a^i ; B", we assume that the node "i" must send some synchronization messages to the *Starting Places* of "B" after "a^i" is executed, and that the nodes belonging to the *Starting Places* of "B" must receive these synchronization messages before any event of "B" is executed. For example, for the behavior expression "a^1 ; b^2 ; c^1 ; exit", we derive the following protocol specifications (here, "m$_1$" and "m$_2$" represent the synchronization messages).

Node 1 : a^1; s$_2$(m$_1$); r$_2$(m$_2$); c^1; exit

Node 2 : r$_1$(m$_1$); b^2; s$_1$(m$_2$); exit



process A := (read1 ; push2 ; A >> pop2 ; write3 ;exit)
[] read1 ; write3 ; exit endproc

Node #	SP	EP	AP
7	{1}	{3}	{1,2,3}
33	{1}	{3}	{1,3}
34	{1}	{3}	{1,3}
37	{3}	{3}	{3}

Fig. 3: Syntax Tree of Process A and Attribute Evaluation

For a behavior expression of the form "B_1 >> B_2", we assume that the nodes belonging to the *Ending Places* of "B_1" must send some synchronization messages to the *Starting Places* of "B_2" after the last event of "B_1" is executed, and that the nodes belonging to the *Starting Places* of "B_2" must receive these synchronization messages before any event of "B_2" is executed.

(2) Non-deterministic choice of alternatives "[]"

Suppose that the behavior expression "a^1 ; b^2 ; c^3 ; exit [] d^1 ; e^3 ; exit" is given. If the synchronization messages described in the above (1) are added to this behavior expression, then the following protocol specifications are obtained.

Node 1 : a^1; $s_2(m_1)$; exit
[] d^1; $s_3(m_3)$; exit
Node 2 : $r_1(m_1)$; b^2; $s_3(m_2)$; exit
[] exit
Node 3 : $r_2(m_2)$; c^3; exit
[] $r_1(m_3)$; e^3; exit

There is no events in the right side of "[]" of the behavior expression for the node 2 . Therefore, an empty alternative of the form "B [] exit" is obtained as the protocol specification of the node 2. If the right side of "[]" is chosen at the node 1 by executing the event "d^1", the node 2 cannot know it. Then, a wrong temporal ordering of the execution of events may occur. This problem occurs when the *All Places* for the behavior expressions of the both sides of "[]" are different. Therefore, we assume that the node executing the first event of any alternative must send synchronization messages to all nodes of the choice expressions which do not participate in the alternative. For the above behavior expression "a^1 ; b^2 ; c^3 ; exit [] d^1 ; e^3 ; exit", we derive the following protocol specifications.

Node 1 : a^1; $s_2(m_1)$; exit
[] d^1; $s_3(m_3)$; *$s_2(m_4)$*; exit
Node 2 : $r_1(m_1)$; b^2; $s_3(m_2)$; exit
[] *$r_1(m_4)$*; exit
Node 3 : $r_2(m_2)$; c^3; exit
[] $r_1(m_3)$; e^3; exit

Here, "*m_4*" is the synchronization messages to inform that the right side of "[]" is chosen.

(3) Process invocation

Let us consider the process A described in Section 2.3. Since the process A may be invoked recursively, it defines the sequence (read^1; push^2)^n; read^1; write^3; (pop^2;

write^3)^n for some $n \geq 0$. If the process A is invoked and the left side of choice operator "[]" is chosen, then a new instance of A, say A', is activated. Again, if the left side of "[]" is chosen, then another instance of A, say A", is activated. Suppose now that the right side of "[]" is chosen for this new instance A", then the process A" will terminate with the execution of the sequence "read^1; write^3; exit". After A" terminates, the sequence "pop^2; write^3" will be executed and A' will also terminate. Then, the process A will be reactivated, and the sequence "pop^2; write^3" will be again executed (see Fig. 2).

It is natural to assume that all nodes in a process should synchronize whenever the process is activated. Therefore, for a behavior expression of the form "a^i; P", we assume that after "a^i" is executed, the node "i" must send some synchronization messages to the *All Places* of "P". In the protocol specifications described in Section 3.1, the node 2 sends synchronization messages to the nodes 1 and 3 after "push^2" is executed (see Fig. 2). Here, if a node "p" does not belong the *All Places* of "P", then the process identifier "P" is replaced by "exit" in the derived protocol specification for the node "p" ("exit" is an event representing the successful termination of a process, and it has no observational effects).

Table 2: Attribute Tp

	Attribute Tp		
(1)	T_p(Process_def)	:=	"process" Proc_Id ":=" T_p(e) "endproc"
(2)	T_p(e_1)	:=	T_p(Par) ">>" **Synch_Left**$_p$(Par,e_2) ">>" **Synch_Right**$_p$(Par,e_2) ">>" T_p(e_2)
(3)	T_p(e)	:=	T_p(Par)
(4)	T_p(Par_1)	:=	T_p(Choice) "|[" **Select**$_p$(event_subset) "]|" T_p(Par_2)
(5)	T_p(Par_1)	:=	T_p(Choice) "|||" T_p(Par_2)
(6)	T_p(Par)	:=	T_p(Choice)
(7)	T_p($Choice_1$)	:=	"(" T_p(Seq) ">>" **Alternative**$_p$(Seq,$Choice_2$) ")" "[]" "(" T_p($Choice_2$) ">>" **Alternative**$_p$($Choice_2$,Seq) ")"
(8)	T_p(choice)	:=	T_p(Seq)
(9)	T_p(Seq_1)	:=	**Proj**$_p$(Event_Id) ";" "(" **Synch_Left**$_p$(Event_Id,Seq_2) ">>" **Synch_Right**$_p$(Event_Id,Seq_2) ">>" T_p(Seq_2) ")"
(10)	T_p(Seq)	:=	**Proj**$_p$("Event_Id") "; exit"
(11)	T_p(Seq)	:=	**Proj**$_p$(Event_Id) ";" "(" **Proc_SynchL**$_p$(Event_Id,Proc_Id) ">>" **Proc_SynchR**$_p$(Event_Id,Proc_Id) ">>" **Proc_Proj**$_p$(Proc_Id) ")"
(12)	T_p(Seq)	:=	"(" T_p(e) ")"

Table 3: Functions Used in Attribute T_p

Synch_Left$_p$ (e_1,e_2)	:=	if (p ∈ EP(e_1)) then
		send ((SP(e_2) - {p}),N(e_1))
		else "empty" endif

Synch_Right$_p$ (e_1,e_2)	:=	if (p ∈ SP(e_2)) then
		receive ((EP(e_1) - {p}), N(e_1))
		else "empty" endif

Proc_SynchL$_p$ (e_1,e_2)	:=	if (p ∈ EP(e_1)) then
		send ((AP(e_2) - {p}), N(e_1))
		else "empty" endif

Proc_SynchR$_p$ (e_1,e_2)	:=	if (p ∈ AP(e_2)) then
		receive ((EP(e_1) - {p}), N(e_1))
		else "empty" endif

Select$_p$ (set)	:=	if set = { } then { }
		else if (set = {e}∪set2 and place(e) = p) then
		{e} ∪ **Select**$_p$(set2)
		else **Select**$_p$(set2) endif
		endif

Proj$_p$(e)	:=	if (p = place(e)) then e else "empty" endif
Proc_Proj$_p$(e)	:=	if (p ∈ AP(e)) then e else "exit" endif

Alternative$_p$ (e_1,e_2)	:=	if (p ∈ SP(e_1)) then
		send ((AP(e_2)-AP(e_1)),N(e_1))
		else if (p ∈ (AP(e_2) - AP(e_1))) then
		receive (SP(e_1),N(e_1))
		else "empty" endif
		endif

send (P,N)	:=	if P = { } then "empty"
		else if P = {i,j,..,k} then "(" s_i(N) ";exit" "||| .. |||" s_k(N) ";exit)" endif
		endif

receive(P,N)	:=	if P = { } then "empty"
		else if P = {i,j,..,k} then "(r_i(N) ";exit" "||| ... |||" r_k(N) ";exit)" endif
		endif

3.2.3 Restrictions for Derivation

In this paper, we treat only the service specifications satisfying the following restrictions.

[Restrictions]

(R1) For each behavior expression of the form "B_1 [] B_2", all starting interactions of "B_1" and all starting interactions of "B_2" must be associated with the same node "q". That is, SP(B_1)=SP(B_1)={q} must hold.

(R2) For each behavior expression of the form "B_1 [] B_2", the set of *Ending Places* of "B_1" and "B_2" must be the same.

(R3) For each behavior expression of the form "B_1 ||| B_2" or "B_1 |[g_1,..,g_n]| B_2", B_1 and B_2 must not invoke the same process. That is, if a process P is invoked in B_1, then the process P must not be invoked in B_2.

Restriction R1 simplifies the decision of which alternative should be selected. Restriction R2 and R3 are introduced in order to simplify the derivation algorithm described in Section 3.3. For example, if R3 does not hold, then the same processes P may be invoked in parallel and the same events a^i in P may be executed simultaneously. For such a case, the synchronization messages sent after the events a^i are executed must be different. This lets the derivation algorithm more complex.

3.3 A Derivation Algorithm

In this section, we propose a derivation algorithm. The algorithm is executed as follows :

Step 1: Construct the syntax tree *Tree*(P_k) of each process definition "P_k := B" in a given service specification L=<P_1,P_2,...,P_n> using the production rules in Table 1.

Step 2: Calculate the attributes SP, EP and AP at each node of the trees *Tree*(P_1), *Tree*(P_2) and *Tree*(P_n) using the attribute evaluation rules in Table 1.

Step 3: For each node "p" in the distributed system, using the attribute evaluation rule for the attribute "T_p" which are defined in Tables 2 and 3, calculate the attribute T_p at each node of the trees *Tree*(P_1), *Tree*(P_2) and *Tree*(P_n).

Let *Pspec*(P_k,p) denote the value of the attribute T_p at the root node of *Tree*(P_k). Then, the protocol specification "L_p" for a node "p" is defined as follows :

L_p=<*Pspec*(P_1,p),*Pspec*(P_2,p),..,*Pspec*(P_n,p)>

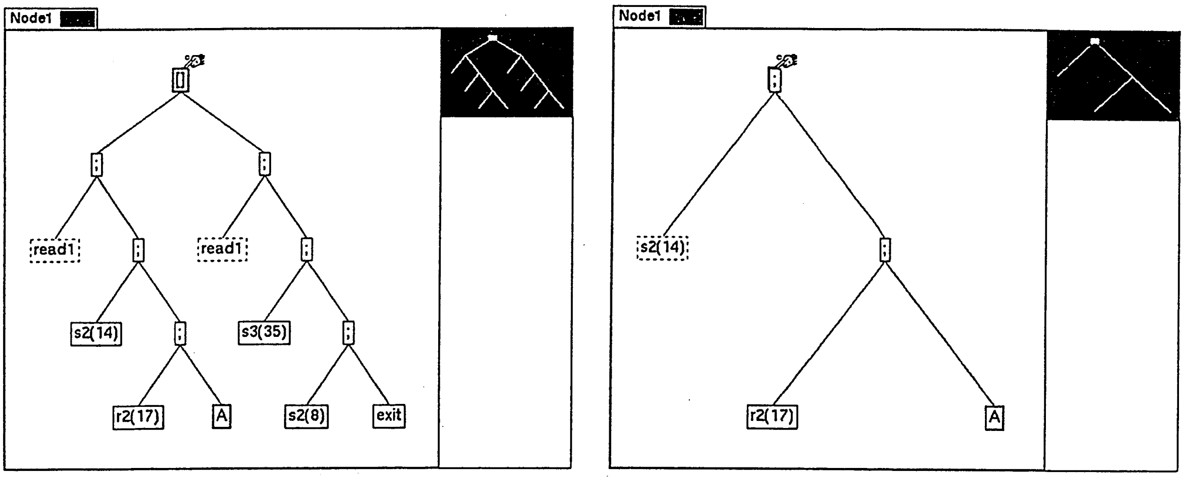
Since the attributes SP, EP, AP and T_p are all the synthesized attributes, the values of the attributes are calculated from the leaf nodes to the root node. The attribute T_p in Step 3 is defined based on the idea described in Section 3.2.2. For example, the attribute evaluation rule (9) in Table 2 represents a derivation algorithm for expressions of the form "a^i ; B". The function **Synch_Left$_p$** represents that if the node "p" belongs to EP(a^i), that is, if "p=i" holds, then the node "p" must send some synchronization messages to the all nodes belonging to SP(B) (see Table 3). The function **Synch_Right$_p$** represents that if the node "p" belongs to SP(B), then it must receive the synchronization message from the node "i" (see Table 3).

In general, the different synchronization messages must be used for the different synchronizations. For example, in the protocol specifications in Section 3.1, different integers such as 8, 14, and 17 are used as the synchronization messages. We may say that the synchronization is defined between the nonterminal symbols in the syntax tree for each process definition of a given service specification. Therefore, we give the node number (integer) "N(e)" to each node "e" of the syntax tree, and use the node numbers as the synchronization messages (see Table 3). By using the above derivation algorithm, the protocol specifications in Section 3.1 are derived from the service specification in Section 2.3. The node numbers in Fig. 3 are used as the synchronization messages for this example.

We have developed the program which derives the protocol specifications from a given service specification(20). By using this program, the protocol specifications are derived automatically.

4. Simulator for Execution of Protocol Specifications

In order to observe the execution processes of LOTOS programs, some LOTOS simulators have been developed(4,22-24). These simulators can simulate the execution of service specifications written in LOTOS. Our LOTOS simulator, *PROSPEX*, can also simulate the execution of protocol specifications(24). If N protocol specifications are given, then N *PROSPEX* are used to simulate them in parallel. We use each *PROSPEX* interactively. *PROSPEX* reads a behavior expression "B" written in LOTOS and shows which events are executable for "B". The user chooses one executable event "e" from the candidates which *PROSPEX* shows. Then, *PROSPEX* executes the event "e" and calculates which events are executable after "e" is executed. The simulation is done by repeating these operations. In *PROSPEX*, the sending/receiving interactions can be executed automatically without interactions from the user.



(a) (b)

Fig. 4: Execution Processes of LOTOS Simulator *PROSPEX*

PROSPEX is executed on UNIX workstations, and it shows these execution processes graphically on X-window. For example, suppose that the protocol specification L_1 in Section 3.1 is given. *PROSPEX* draws the syntax tree of the behavior expression of the process "A" (see Fig. 4(a)) on X-window. Each leaf corresponds to either an event, an sending/receiving interaction or a process. The dotted rectangles corresponds to executable events. In Fig. 4(a), two events "read^1" of the both sides of the choice operator "[]" are executable. If the user cricks "read^1" of the left side of "[]", then the event is executed and the syntax tree in Fig. 4(a) is replaced by that in Fig. 4(b). This shows that the event "read^1" is executed and a new behavior expression, say B', is obtained. For B', the sending interaction $s_2(14)$ and receiving interaction $r_2(17)$ are executed automatically without interactions from the user. Then, the behavior expression B' is replaced by the process A. Since some events in the process A are executable, the node corresponding to A is replaced by the syntax tree of the behavior expression of A automatically. The replaced syntax tree is the same as that in Fig. 4(a). Even if the size of a syntax tree becomes large, *PROSPEX* calculates a suitable size for drawing the tree on the given window and draws it.

5. Conclusion

In this paper, a derivation algorithm of protocol specifications from a service specification is introduced. In general, the protocol specifications derived from a service specification are not simple even if a very simple service specification such as the example described in Section 2.3 is given. Therefore, our approach to derive protocol specifications from a service specification is a good approach to design distributed systems. For service specifications written in Full LOTOS, the derivation algorithm in Ref. (20) is useful. The formal proof of the correctness of the derivation algorithm is a future work.

References

(1) ISO : "Information Processing System - Open Systems Interconnection -LOTOS- A Formal Description Technique based on the Temporal Ordering of Observational Behaviour", IS 8807, 1989.

(2) P. H. J. van Eijk, C.A. Vissers and M. Diaz : "The Formal Description Technique LOTOS", North Holland, 1989.

(3) T. Bolognesi and E. Brinskma : "Introduction to the ISO Specification Language LOTOS", Computer Networks and ISDN Systems, Vol. 14, No. 1, pp 25-59, 1987.

(4) K. Takahashi, H. Kaminaga and N. Shiratori : "LOTOS Features with Survey of Their Support Processing Systems", J. IPS of Japan, Vol.31, No. 1, pp.35-46, 1990 (in Japanese).

(5) ISO : "Information Processing System - Open Systems Interconnection - Basic Reference Model", IS 7498, 1984.

(6) K. Ohmaki and K. Futatugi : "Early Experience with a Formal Description Technique : LOTOS", J. IPS of Japan, Vol. 31, No. 10, pp.1400-1413, 1990 (in Japanese).

(7) C. Vissers and L. Logrippo : "The Importance of the Concept of Service in the Design of Data Communications Protocols", Proceedings of the Fifth IFIP Workshop on Protocol Specification, Verification and Testing, North Holland, pp.3-17, 1985.

(8) R. Probert and K. Saleh : "Synthesis of Communication Protocols : Survey and Assessment", IEEE Trans. Comput., Vol. 40, No. 4, pp.468-476, 1991.

(9) P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan and D. Brand : "Towards Analyzing and Synthesizing Protocols", IEEE Trans. Commun., Vol. COM-28, No.4, pp.651-661, 1980.

(10) C.V. Ramamoorthy, S. T. Dong and Y. Usuda : "An Implementation of an Automated Protocol Synthesizer (APS) and its Application to the X.21 Protocol", IEEE Trans. Software Eng., Vol. SE-11, No.9, pp. 886-908, 1985.

(11) C.V., Ramamoorthy, Y. Yaw, R. Aggarwal and J. Song : "Synthesis of Two-Party Error-Recoverable Protocols", Proceedings of the ACM SIGCOMM '86 Symposium, pp.227-235, 1986.

(12) P. Merlin and G. von Bochmann : "On the Construction of Submodule Specifications and Communication Protocols", ACM Trans. Program. Lang. & Syst., No.1, pp.1-25, 1983.

(13) M. Gouda and Y. Yu : "Synthesis of Communicating Finite State Machines with Guaranteed Progress", IEEE Trans. Commun., Vol. COM-32, No. 7, pp.779-788, 1984.

(14) P. M. Chu and M.T. Liu : "Protocol Synthesis in a State Transition Model", Proceedings IEEE COMPSAC' 88, pp. 505-512, 1988.

(15) G. von Bochmann and R. Gotzhein : "Deriving Protocol Specifications from Service Specifications", Proceedings of the ACM SIGCOMM '86 Symposium, Vermont, USA, pp.148-156, 1986.

(16) T. Higashino, T. Kimoto, K. Taniguchi and M. Mori : "Synthesis of Protocol Machines from Service Specification", Technical Report of IPS of Japan, 88-SF-26-5, 1988 (in Japanese).

(17) R. Gotzhein and G. von Bochmann : "Deriving Protocol Specifications from Service Specifications Including Parameters", ACM Trans. Comput. Syst., Vol. 8, No. 4, pp.253-283, 1990.

(18) F. Khendek, G. von Bochmann and C. Kant : "New results on deriving protocol specifications from services specifications", Proceedings of the ACM SIGCOMM'89, pp.136-145, 1989.

(19) C. Kant, T. Higashino and G. von Bochmann : "Deriving Protocol Specifications from Service Specifications Written in Basic LOTOS", (submitted for publications).

(20) T. Higashino, R. Katou, K. Yasumoto and K. Taniguchi: "Deriving Protocol Specifications from Service Specification Written in LOTOS with Data Parameters", Technical Report of IEICE of Japan, IN91-111, 1991 (in Japanese).

(21) R. Langerak : "Decomposition of Functionality; a Correctness-Preserving LOTOS Transformation", Proceedings of the Tenth International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, North Holland, pp.229-242, 1990.

(22) J. Tretmans : "Hippo : A LOTOS Simulator", The Formal Description Technique LOTOS, North-Holland, pp.391-396, 1989.

(23) R. Guillemot, M. Haj-Hussein and L. Logrippo : "Executing Large LOTOS Specifications", Proceedings of the Eighth International IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, North Holland, pp.399-410, 1988.

(24) K. Yasumoto, T. Higashino and K. Taniguchi : "Execution of Protocol Specifications Written in LOTOS", Technical Report of IEICE of Japan, IN91-112, 1991 (in Japanese).

(25) H. Ehrig and B. Mahr : "Fundamentals of Algebraic Specification 1", EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer-Verlag, 1985.

(26) A. Aho, R. Sethi and J. D. Ullman : "Compilers Principles, Techniques and Tools", Addison-Wesley, 1985.