<table>
<thead>
<tr>
<th>タイトル</th>
<th>Service Specification and Protocol Specifications in LOTOS: Equivalence and Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>著者</td>
<td>Higashino, Teruo; Yasumoto, Keiichi; Taniguchi, Kenichi</td>
</tr>
<tr>
<td>引用</td>
<td>数理解析研究所講究録 (1992), 790: 215-221</td>
</tr>
<tr>
<td>発行日</td>
<td>1992-06</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/2433/82648">http://hdl.handle.net/2433/82648</a></td>
</tr>
<tr>
<td>タイプ</td>
<td>Departmental Bulletin Paper</td>
</tr>
<tr>
<td>テキストバージョン</td>
<td>publisher</td>
</tr>
</tbody>
</table>

Kyoto University
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, unspecificationed 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 E Ecutor) 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 "\(\mid\)"

A behavior expression "\(B_1 \mid B_2\)" 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 "\(\parallel\)"

A behavior expression "\(B_1 \parallel B_2\)" 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 "B1 \[(g1,..,gn)\] B2" represents that both behavior expressions "B1" and "B2" are executable in parallel. The events in "B1" and "B2" belonging to \[(g1,..,gn)\] must be executed as rendezvous interactions. If all events in "B1" and "B2" are contained in \[(g1,..,gn)\], then "B1 \[(g1,..,gn)\] B2" may be described as "B1 \ll B2".

(5) Sequential composition ">>
A behavior expression "B1 >> B2" represents that the behavior expression "B2" is executable after the execution of the behavior expression "B1" is finished successfully.

(6) Disabling operator "\sim"
Disabling operator \sim represents the interruption of a particular sequence of events by a disabling event. A behavior expression "B1 >> B2" represents that the behavior expression "B1" is executable until an event "d" of the behavior expression "B2" is executed. If "d" is executed, then only the events in "B2" 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 behavior specifications and derived protocol specifications.

Suppose that a LOTOS program \(L\) consists of the tuple \(L=\langle P_1, P_2, \ldots, P_n \rangle\) of \(n\) processes \(P_1, P_2, \ldots, P_n\), 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 "Identifier:Node" where "Identifier" denotes the service primitive itself and "Node" denotes the node name at which the interaction takes place. For example, "a2\;" 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\{(m)\}" which means the sending of the message "m" to the node "i".
- an interaction of receiving message: It is written as "r\{(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 "read1" which is a service primitive interaction reading a file from the file. At the node 3, we can only execute "write3" 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 "push2" or "pop2". "Push2" inserts the last receiving element in the local stack.

"Pop2" 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:

\[
L=\langle A \rangle \\
\text{process } A := (\text{read}^1; \text{push}^2; A >\text{pop}^2; \text{write}^3; \text{exit}) \]
\[
[] \text{read}^1; \text{write}^3; \text{exit endproc}
\]

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:
\[
L_1 = \langle A \rangle \\
\text{process } A := \text{read}^1; s_2(14); r_2(17); A \\
[] \text{read}^1; s_3(35); s_2(8); \text{exit endproc}
\]

Node 2:
\[
L_2 = \langle A \rangle \\
\text{process } A := (r_1(14); \text{push}^2; (s_1(17); \text{exit} \ll s_3(17); \text{exit}) \\
>\text{A >> r}_3(11); \text{pop}^2; s_3(25); \text{exit}) \\
[] r_1(8); \text{exit endproc}
\]

Node 3:
\[
L_3 = \langle A \rangle \\
\text{process } A := (r_2(17); A >\text{sg}(11); r_2(25); \text{write}^3; \text{exit}) \\
[] r_1(35); \text{write}^3; \text{exit endproc}
\]

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

\[
L_2 = \langle A \rangle \\
\text{process } A := (r_1(14); \text{push}^2; (s_1(17); \text{exit} \ll s_3(17); \text{exit}) \\
>\text{A >> r}_3(11); \text{pop}^2; s_3(25); \text{exit}) \\
[] r_1(8); \text{exit endproc}
\]

Fig. 2: An Execution Process of Protocol Specifications
Table 1: Syntax of Specification Language and Attribute Evaluation Rules

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Production Rules</th>
<th>Attribute SP (Starting Places)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td><code>Process_def -&gt; process Proc_Id := e endproc</code></td>
<td>$SP(\text{Process_def}) = SP(e)$</td>
</tr>
<tr>
<td>(2)</td>
<td><code>e1 = e2</code></td>
<td>$SP(e1) = SP(\text{Par})$</td>
</tr>
<tr>
<td>(3)</td>
<td><code>e -&gt; Par</code></td>
<td>$SP(e) = SP(\text{Par})$</td>
</tr>
<tr>
<td>(4)</td>
<td><code>Par1 -&gt; Choice [\text{event_subst}] Par2</code></td>
<td>$SP(\text{Par1}) = SP(\text{Choice}) \cup SP(\text{Par2})$</td>
</tr>
<tr>
<td>(5)</td>
<td><code>Par1 -&gt; Choice \text{ iii } Par2</code></td>
<td>$SP(\text{Par1}) = SP(\text{Choice}) \cup SP(\text{Par2})$</td>
</tr>
<tr>
<td>(6)</td>
<td><code>Par -&gt; Choice</code></td>
<td>$SP(\text{Par}) = SP(\text{Choice})$</td>
</tr>
<tr>
<td>(7)</td>
<td><code>Choice1 -&gt; Seq \{} Choice2 \}</code></td>
<td>$SP(\text{Choice1}) = SP(\text{Seq}) = SP(\text{Choice2})$</td>
</tr>
<tr>
<td>(8)</td>
<td><code>Choice -&gt; Seq</code></td>
<td>$SP(\text{Choice}) = SP(\text{Seq})$</td>
</tr>
<tr>
<td>(9)</td>
<td><code>Seq -&gt; Event_Id : Seq2</code></td>
<td>$SP(\text{Seq}) = {\text{place(Event_Id)}}$</td>
</tr>
<tr>
<td>(10)</td>
<td><code>Seq -&gt; Event_Id ; exit</code></td>
<td>$SP(\text{Seq}) = {\text{place(Event_Id)}}$</td>
</tr>
<tr>
<td>(11)</td>
<td><code>Seq -&gt; Event_Id : Proc_Id</code></td>
<td>$SP(\text{Seq}) = {\text{place(Event_Id)}}$</td>
</tr>
<tr>
<td>(12)</td>
<td><code>Seq -&gt; (e)</code></td>
<td>$SP(\text{Seq}) = SP(e)$</td>
</tr>
</tbody>
</table>

Attributes EP (Ending Places) and AP (All Places)

<table>
<thead>
<tr>
<th>EP</th>
<th>AP</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$EP(\text{Process_def}) = EP(e)$</td>
</tr>
<tr>
<td>(2)</td>
<td>$EP(e_1) = EP(e_2)$</td>
</tr>
<tr>
<td>(3)</td>
<td>$EP(e) = EP(\text{Par})$</td>
</tr>
<tr>
<td>(4)</td>
<td>$EP(\text{Par1}) = EP(\text{Choice}) \cup EP(\text{Par2})$</td>
</tr>
<tr>
<td>(5)</td>
<td>$EP(\text{Par1}) = EP(\text{Choice}) \cup EP(\text{Par2})$</td>
</tr>
<tr>
<td>(6)</td>
<td>$EP(\text{Par}) = AP(\text{Choice})$</td>
</tr>
<tr>
<td>(7)</td>
<td>$EP(\text{Choice1}) = EP(\text{Seq}) = EP(\text{Choice2})$</td>
</tr>
<tr>
<td>(8)</td>
<td>$EP(\text{Choice}) = EP(\text{Seq})$</td>
</tr>
<tr>
<td>(9)</td>
<td>$EP(\text{Seq}) = EP(\text{Seq})$</td>
</tr>
<tr>
<td>(10)</td>
<td>$EP(\text{Seq}) = {\text{place(Event_Id)}}$</td>
</tr>
<tr>
<td>(11)</td>
<td>$EP(\text{Seq}) = {\text{place(Event_Id)}}$</td>
</tr>
<tr>
<td>(12)</td>
<td>$EP(\text{Seq}) = EP(e)$</td>
</tr>
</tbody>
</table>

(Here, place(Identifier \text{Node}) = \text{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").

- **\text{SP}(x)**: The set of nodes where the first events of $\exp(x)$ are executed. It is called \textit{Starting Places} of the nonterminal symbol "x".
- **\text{EP}(x)**: The set of nodes where the last events of $\exp(x)$ are executed. It is called \textit{Ending Places} of "x".
- **\text{AP}(x)**: The set of all nodes where the events of $\exp(x)$ are executed. It is called \textit{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 $\text{SP}(x)$, $\text{EP}(x)$, and $\text{AP}(x)$ is a nonterminal symbol, we may use $\exp(x)$ instead of "x". That is, we may use $\text{SP}(\exp(x))$ instead of $\text{SP}(x)$ if there is no confusion.

We give the attributes $\text{SP}(x)$, $\text{EP}(x)$, and $\text{AP}(x)$ not only for the nonterminal symbols but also for the leaves corresponding to event identifiers and process identifiers. If "Event_Id" is "Identifiers\text{Node}" then we define $\text{SP(\text{Event_Id}) = EP(\text{Event_Id}) = AP(\text{Event_Id}) = \{\text{Node}\}$.

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 "$\text{SP}(A) = \text{SP}(A) \cup X$" holds, then "$\text{SP}(A) = X$" is obtained as the solution. For the process A in Fig. 3, the attribute $\text{SP}(A)$, $\text{EP}(A)$, and $\text{AP}(A)$ are treated as variables. We find the equations $\text{SP}(A) = \{1,2\}$, $\text{EP}(A) = \{1\}$, and $\text{AP}(A) = \{1,2\}$. Therefore, the solutions are $\text{SP}(A) = \{1\}$, $\text{EP}(A) = \{1\}$, and $\text{AP}(A) = \{1,2\}$. 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 "\text{a}_i : \text{B}", we assume that the node "i" must send some synchronization messages to the \textit{Starting Places} of "B" after \text{a}_i is executed, and that the nodes belonging to the \textit{Starting Places} of "B" must receive these synchronization messages before any event of "B" is executed. For example, for the behavior expression "\text{a}_1 : \text{b}_2 : \text{c}_1 : \text{exit}", we derive the following protocol specifications (here, \text{m}^1 and \text{m}^2 represent the synchronization messages).

Node 1: \text{a}^1; \text{m}_1^1; \text{m}_2^1; \text{c}^1; \text{exit}

Node 2: \text{a}^1; \text{m}_1^2; \text{m}_2^2; \text{c}^1; \text{exit}
For a behavior expression of the form "B₁ >> B₂", we assume that the nodes belonging to the Ending Places of "B₁" must send some synchronization messages to the Starting Places of "B₂" after the last event of "B₁" is executed, and that the nodes belonging to the Starting Places of "B₂" must receive these synchronization messages before any event of "B₂" is executed.

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

Suppose that the behavior expression "a₁; b²; c³; exit [ ] d¹; e³; 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: a₁; s₂(m₁); exit
[ ] d¹; s₃(m₃); exit

Node 2: r₁(m₁); b²; s₃(m₂); exit
[ ] exit

Node 3: r₂(m₂); c³; exit
[ ] r₁(m₃); c³; 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 "B₁ 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 "d¹", 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 "a₁; b²; c³; exit [ ] d¹; e³; exit", we derive the following protocol specifications.

Node 1: a₁; s₂(m₁); exit
[ ] d¹; s₃(m₃); s₂(m₄); exit

Node 2: r₁(m₁); b²; s₃(m₂); exit
[ ] r₁(m₄); exit

Node 3: r₂(m₂); c³; exit
[ ] r₁(m₃); c³; exit

Here, "m₄" 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 (read¹; push₂)ₓ; read¹; write³; (pop²;
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 "\(a\); \(P\)", we assume that after "\(a\)" 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

<table>
<thead>
<tr>
<th></th>
<th>Attribute Tp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T_p(\text{Process_def}) \Rightarrow \text{&quot;process&quot; } \text{Proc_Id} = \text{&quot;} \text{T_p(e)} \text{&quot; endproc} )</td>
</tr>
<tr>
<td>2</td>
<td>(T_p(e_1) \Rightarrow T_p(\text{Par}) \Rightarrow \text{&quot;} \text{Synch_Leftp(Par,e_2)} \Rightarrow \text{&quot;} \text{Synch_Rightp(Par,e_2)} \Rightarrow \text{&quot;} \text{T_p(e_2)} )</td>
</tr>
<tr>
<td>3</td>
<td>(T_p(e) \Rightarrow T_p(\text{Par}) )</td>
</tr>
<tr>
<td>4</td>
<td>(T_p(\text{Par}) \Rightarrow T_p(\text{Choice}) \Rightarrow \text{&quot;} \text{Selectp(event_subset)} \Rightarrow \text{&quot;} \text{T_p(Par)} )</td>
</tr>
<tr>
<td>5</td>
<td>(T_p(\text{Par}) \Rightarrow T_p(\text{Choice}) \Rightarrow \text{&quot;} \text{ILL} \Rightarrow \text{&quot;} \text{T_p(Par)} )</td>
</tr>
<tr>
<td>6</td>
<td>(T_p(\text{Par}) \Rightarrow T_p(\text{Choice}) )</td>
</tr>
<tr>
<td>7</td>
<td>(T_p(\text{Choice}) \Rightarrow \text{&quot;} \text{&quot;} \text{TP(Par)} \Rightarrow \text{&quot;} \text{Alternativep(Par,Choice,e_2)} \Rightarrow \text{&quot;} \text{&quot;} \text{TP(Par)} )</td>
</tr>
<tr>
<td>8</td>
<td>(T_p(\text{choice}) \Rightarrow \text{&quot;} \text{T_p(Par)} \Rightarrow \text{&quot;} \text{T_p(Par)} )</td>
</tr>
<tr>
<td>9</td>
<td>(T_p(\text{Seq1}) \Rightarrow \text{Projp(Event_Id)} \Rightarrow \text{&quot;} \text{&quot;} \text{Synch_Leftp(Event_Id,Seq)} \Rightarrow \text{&quot;} \text{Synch_Rightp(Event_Id,Seq)} \Rightarrow \text{&quot;} \text{T_p(Seq)} )</td>
</tr>
<tr>
<td>10</td>
<td>(T_p(\text{Seq}) \Rightarrow \text{Projp(Event_Id)} \Rightarrow \text{&quot;} \text{&quot;} \text{T_p(Seq)} )</td>
</tr>
<tr>
<td>11</td>
<td>(T_p(\text{Seq}) \Rightarrow \text{Projp(Event_Id)} \Rightarrow \text{&quot;} \text{&quot;} \text{T_p(Seq)} )</td>
</tr>
<tr>
<td>12</td>
<td>(T_p(\text{Seq}) \Rightarrow \text{&quot;} \text{T_p(e)} \Rightarrow \text{&quot;} )</td>
</tr>
</tbody>
</table>

Table 3: Functions Used in Attribute Tp

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{Synch_Leftp(e_1,e_2)} )</td>
<td>if (p \in \text{EP(e_1)}) then send (((\text{SP(e_2)}) (\cdot) (p))(\cdot)(N(e_1))) else &quot;empty&quot; endif</td>
</tr>
<tr>
<td>(\text{Synch_Rightp(e_1,e_2)} )</td>
<td>if (p \in \text{SP(e_2)}) then receive (((\text{EP(e_1)}) (\cdot) (p))(\cdot)(N(e_1))) else &quot;empty&quot; endif</td>
</tr>
<tr>
<td>(\text{Proc_SynchLp(e_1,e_2)} )</td>
<td>if (p \in \text{EP(e_1)}) then send (((\text{AP(e_2)}) (\cdot) (p))(\cdot)(N(e_1))) else &quot;empty&quot; endif</td>
</tr>
<tr>
<td>(\text{Proc_SynchRp(e_1,e_2)} )</td>
<td>if (p \in \text{AP(e_2)}) then receive (((\text{EP(e_1)}) (\cdot) (p))(\cdot)(N(e_1))) else &quot;empty&quot; endif</td>
</tr>
<tr>
<td>(\text{Selectp(set)} )</td>
<td>if set = {} then {} else if ({e} \cup \text{Selectp(set)}) then ({e} \cup \text{Selectp(set)}) endif</td>
</tr>
<tr>
<td>(\text{Projp(e)} )</td>
<td>if (p = \text{place(e)}) then (e) else &quot;empty&quot; endif</td>
</tr>
<tr>
<td>(\text{Proc_Projp(e)} )</td>
<td>if (p \in \text{AP(e)}) then (e) else &quot;exit&quot; endif</td>
</tr>
<tr>
<td>(\text{Alternativep(e_1,e_2)} )</td>
<td>if (p \in \text{SP(e_1)}) then send (((\text{AP(e_2)}) (\cdot) (e))(\cdot)(N(e_1))) else if (p \in \text{AP(e_2)}) then receive (((\text{SP(e_1)}) (\cdot) (N(e_1))) else &quot;empty&quot; endif</td>
</tr>
<tr>
<td>(\text{send(P,N)} )</td>
<td>if (P = {}) then &quot;empty&quot; else if (P = {i_1,...,k}) then &quot;(s_1(N) \cdot \text{exit} \cdot ... \cdot \text{exit} \cdot s_k(N) \cdot \text{exit})&quot; endif</td>
</tr>
<tr>
<td>(\text{receive(P,N)} )</td>
<td>if (P = {}) then &quot;empty&quot; else if (P = {i_1,...,k}) then &quot;(r_1(N) \cdot \text{exit} \cdot ... \cdot \text{exit} \cdot r_k(N) \cdot \text{exit})&quot; endif</td>
</tr>
</tbody>
</table>
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 $B_1 \parallel B_2$, all starting interactions of $B_1$ and all starting interactions of $B_2$ must be associated with the same node $q$. That is, $SP(B_1) = SP(B_2) = \{q\}$ must hold.

(R2) For each behavior expression of the form $B_1 \parallel B_2$, the set of Ending Places of $B_1$ and $B_2$ must be the same.

(R3) For each behavior expression of the form $B_1 \parallel B_2$ or $B_1 \parallel [g_1, \ldots, g_n] B_2$, $B_1$ 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 $L = <P_1, P_2, \ldots, P_n>$ 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 $"Tp"$ which are defined in Tables 2 and 3, calculate the attribute $Ty_p$ at each node of the trees $Tree(P_1)$, $Tree(P_2)$ and $Tree(P_n)$.

Let $Ps(P_k,p)$ denote the value of the attribute $Ty_p$ at the root node of $Tree(P_k)$. Then, the protocol specification "$LP" for a node $p'$ is defined as follows:

$LP = \langle Ps(P_1,p), Ps(P_2,p), \ldots, Ps(P_n,p) \rangle$

Since the attributes $SP$, $EP$, and $Ty_p$ are all the synthesized attributes, the values of the attributes are calculated from the leaf nodes to the root node. The attribute $Ty_p$ 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 $"a_1 ; B"$. The function $Synch_{Left_p}$ represents that if the node "$p" belongs to $EP(a_i)$, that is, if "$p = a_i" holds, then the node "$p" must send some synchronization messages to 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 we 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 the 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 a 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.

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 L1 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” of the both sides of the choice operator "|" are executable. If the user clicks "read” 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” is executed and a new behavior expression, say B’, is obtained. For B’, the sending interaction r2(14) and receiving interaction r2(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