Execution of Distributed Systems described in LOTOS and its Visualization

Keiichi Yasumoto

January 1996

Execution of Distributed Systems described in LOTOS and its Visualization

by

Keiichi Yasumoto

January 1996

Dissertation submitted to the Faculty of the Engineering Science
of Osaka University in partial fulfillment of the requirements for
the degree of a Doctor of Philosophy in Engineering

Abstract

This thesis summarizes the work of the author as a master/doctor student of Osaka University and a research associate of Shiga University on the execution of distributed systems described in LOTOS and its visualization.

The behavior of a distributed system can be specified formally by defining input/output interactions (called *events*) and their temporal ordering offered to the external environment. LOTOS is a formal description language, and it has been standardized within ISO. To design and develop such a distributed system efficiently using formal languages like LOTOS, it is desirable to describe the system specification as a program executed at one node (called *service specification*) in the abstract level, and to derive a tuple of programs (called *protocol entity specifications*) for those nodes automatically, at the next stage.

Generally, in designing phases, the system specifications are frequently modified for debugging and/or improving the systems as the results of the behavior analysis. For the efficient behavior analysis of distributed systems, it is desirable to execute the tuple of protocol entity specifications in a distributed environment and to display the dynamic behavior of the specifications visually.

Final specifications of the system should be converted into efficient object codes for a given target machine. LOTOS specifications contains many operators such as choice, parallel, disabling and synchronization to specify the temporal ordering of events among processes. So, it is complicated to implement such LOTOS specifications in the procedural languages such as C manually.

This thesis provides the following three research topics for these purposes.

First, for a given service specification with data parameters and all basic operators in LOTOS, and an assignment of each gate to a node, an algorithm to derive correct protocol entity specifications is proposed. Although the proposed derivation algorithm still imposes some restrictions such that the alternative events must belong to the same node as well as the existing algorithms, there were no algorithms dealing with such a wide class. The algorithm assumes that each communication message is exchanged asynchronously through a reliable communication channel like a FIFO queue between any two nodes. In order to derive protocol entity specifications, communication events may have to be calculated from the temporal ordering of events and data dependency among distributed nodes. A trivial solution is that each node broadcasts the information about each event execution and input data to all the other nodes. However, this technique requires so many message exchanges and much time to execute such communications. It is desirable to reduce the number of the communication events, especially in the real-time systems. In the proposed algorithm, to derive only the necessary communication events, several functions are defined for the syntax tree of the given service specification. The functions calculate the sets of nodes which can execute first and last executable events, respectively, and the set of the pairs of the data name and node using the data. To calculate such sets easily, the derivation algorithm is described in an attribute grammar.

For example, the derivation algorithm has been applied to the ISPW6 problem, an open problem for modeling concurrent software processes consisting of several engineers. From the service specification of this problem, the protocol entity specifications have been derived in the practical time.

Secondly, a simulator which executes each protocol entity specification and displays its dynamic behavior visually, is proposed. For analysis of the distributed systems, the simulator executes the communication events in a protocol entity specification and communicates with other nodes via the network. N simulators execute n protocol entity specifications at the nodes so that the whole behavior of the service specification is implemented in parallel. For facilitating the dynamic behavior of LOTOS specifications with its structural information, it is useful to display the syntax tree of the current behavior expression visually and to update the syntax tree step by step at every event execution. The proposed simulator uses a fast layout algorithm to depict such tree structures rapidly. The simulator can also trace what event is executed in the service specification while the tuple of the protocol entity specifications are executed.

Thirdly, an efficient implementation method of protocol entity specifications using a multi-thread mechanism on a given target machine is proposed. There are several existing implementation methods to generate efficient object codes by restricting the target class such that the tuple of synchronizing processes is statically decided. In the proposed implementation method, we deal with a wider class which includes that the tuple of synchronizing processes is dynamically decided, by using a shared data area to exchange the dynamic information among the threads. In the method, each sequentially executable sub-expression is mapped to a thread in the object code, and all threads in the object code are created concurrently. All these autonomous threads access the same shared data area (called *the control area*) so that they execute events in the order specified in the specification. To implement the hierarchically specified operators of LOTOS behavior expressions, a control area consists of the structured data areas where each area is used to implement the corresponding operator. The temporal ordering of events between two threads specified by choice, disabling, synchronization and so on, is implemented by accessing the corresponding area in the control area. To avoid the control area enlarging, the obsolete parts of the area are dynamically removed and a new area is added.

To monitor the dynamic behavior of the real-time systems, a visualization technique for the derived object codes is proposed. To visualize a system specification without modifying it, we combine a system specification and its visualization scenario with synchronization operators for the events to be visualized. A tuple of the original specification and its scenario is converted into the multi-threaded object code by the compiler.

Some experimental results have shown that the compiler can generate more efficient object codes than other LOTOS compilers with respect to the parallel and choice execution of events. By visualizing an example "five dining philosophers", it is shown that the dynamic behavior of the concurrent processes can be animated enough fast.

List of Major Publications

(1) Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K. : PROSPEX: A Graphical LOTOS Simulator for Protocol Specifications with N Nodes, *IEICE Trans. Commun.*, Vol.E75-B, No.10, pp. 1015 – 1023, Oct. 1992.

(2) Yasumoto, K., Higashino, T. and Taniguchi, K. : Software Process Description Using LOTOS and Its Enaction, Proc. of 16th IEEE Int. Conf. on Software Engineering (ICSE-16), pp. 169 – 178, May 1994.

(3) Yasumoto, K., Higashino, T. and Taniguchi, K.: Software Process Description in LOTOS And Its Execution, *Journal of Japan Society for Software Science and Technology*, Vol.12, No.1, pp. 16 – 30, Jan. 1995 (in Japanese).

(4) Yasumoto, K., Higashino, T., Abe, K., Matsuura, T. and Taniguchi, K. : A LOTOS Compiler Generating Multi-threaded Object Codes, Proc. of 8th IFIP Int. Conf. on Formal Description Techniques (FORTE'95), Chapman & Hall, pp. 271 – 286, Oct. 1995.

(5) Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K. : Protocol Visualization using LOTOS Multi-Rendezvous Mechanism, Proc. of 1995 IEEE Int. Conf. on Network Protocols (ICNP-95), pp. 118 – 125, Nov. 1995.

(6) Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K. : Visualizing Dynamic Behavior of LOTOS specifications using Multi-rendezvous Mechanism (in Japanese) (submitted for publications as a journal paper of IPSJ: current status: conditionally accepted).

Acknowledgments

This work could be achieved owing to a great deal of helps of many individuals.

First, I would like to thank my supervisor Professor Kenichi Taniguchi of Osaka University, for his continuous support, encouragement and guidance of the work.

I'm very grateful to Professor Mamoru Fujii and Professor Kenichi Hagihara for their invaluable comments and helpful suggestions concerning this thesis. I'm also very grateful to Professor Nobuki Tokura, Professor Seishi Nishikawa, Professor Hideo Miyahara, Professor Toru Kikuno, and Professor Katsuro Inoue for their valuable comments on this thesis.

I would like to express my sincere gratitude to Associate Professor Teruo Higashino of Osaka University for his adequate guidance, valuable suggestions and discussions throughout this work. This work could not be achieved without his continuous support, encouragement and guidance.

I also wish to thank to Professor Toshio Matsuura of Osaka City University for his technical supports, valuable comments and discussions about the work. The tools which has been designed and developed by his research group, such as a visual tree editor, a portable thread mechanism, and an animation server are used to develop the LOTOS simulator, the compiler and the visualization tool described in Chapter 4 and Chapter 5 in this thesis.

I'd like to express my thanks to Professor Koji Torii, Associate Professor Hajimu Iida of Nara Institute of Science and Technology, and Assistant Professor Masahiro Higuchi of Osaka University for their helpful comments and suggestions.

I'm grateful to Professor Masaaki Mori, Professor Yoshiki Katsuyama and Associate Professor Shinichi Taniguchi of Shiga University for their helpful suggestions.

Many of the courses that I have taken during my graduate career have been helpful to prepare this thesis. I would like to acknowledge the guidance of Professors Toshinobu Kashiwabara, Masaru Sudo, and Akihiro Hashimoto.

I also wish to thank research associates of Osaka University, Mr. Junji Kitamichi and Dr. Kozo Okano for their helpful suggestions.

I wish to express my special gratitude to Mr. Toru Nakamura, Mr. Kota Abe, Mr. Noriaki Yamashita, Mr. Hiroaki Yoshio, Mr. Takahiro Shiroshima, Ms. Yoshimi Enya, Mr. Kazuhiro Goto and Mr. Motohiro Miyamoto for their works on developing the parts of the systems described in Chapter 4 and Chapter 5.

I'm thankful to Mr. Sumio Morioka, a Ph.D. student of Osaka University for his valuable comments.

Finally, I would like to thank the all members of Taniguchi Laboratory of Osaka University for their helpful advice.

Contents

Chapter 1

Introduction

In recent years, distributed systems consisting of several distributed processors have been popular as the growth of the computer networks. However, the bugs in such systems may cause incredible disasters in human society. So, many techniques to design and develop reliable distributed systems efficiently have been studied. In this thesis, a technique to design and develop reliable distributed systems described in concurrent languages is proposed.

In general, we use the formal models based on the finite state machines (FSM) or process algebras such as CCS [42] to specify distributed systems and/or communication protocols. LOTOS is a formal description language based on CCS, and it has been standardized within ISO since 1989. In this work, LOTOS is chosen as the target language for describing distributed systems.

In LOTOS, a system specification is described by defining input or output interactions (called *events*) and their temporal ordering offered to the external environment of the system. In LOTOS, such events and their temporal ordering are described as the *behavior expression*. LOTOS has useful operators to specify the temporal ordering of events in the behavior expression, such as the alternative, parallel composition between any event sequences, and disabling operator which disables a specified part of the behavior expression by a particular event execution. In LOTOS, whole the behavior expression of the system can be separated into sub modules called *processes*. Processes can be described in the behavior expressions of their parent processes as well as events. Consequently, LOTOS specifications are described as a process consisting of several sub-processes. One of important characteristics of LOTOS is the multi-rendezvous mechanism [31] which enables multiple concurrent processes to execute specified events simultaneously and to exchange data values. Using the above mechanisms, we can describe the specifications of complex distributed systems, simply and clearly.

In order that several processors (nodes) cooperate to execute some meaning event sequences as a service in a distributed environment, each node must exchange communication messages with other nodes as well as execute its events. In designing and developing a high reliable distributed system using LOTOS, it is a useful technique to describe the system specification as a program executed

at one node (called *service specification*) in the abstract level, and to derive a tuple (called *protocol specification*) of programs for those nodes automatically, at the next stage. In a program of each node (called *protocol entity specification*), there must be specified the contents and the temporal ordering of the sending and receiving interactions (called *communication events*) to other nodes for preserving the temporal ordering of events and the distribution of the data parameters between those nodes. Since a number of communication events are needed if choice and parallel operators are specified in the service specification, it is complicated for the designers to specify such communication events in each protocol entity specification manually without mistakes. Automatic derivation of such protocol entity specifications will bring efficient design and development of the high reliable distributed systems.

For such purposes, several researches to derive correct protocol specifications for the various classes of LOTOS have been studied [19, 24, 36, 37]. The techniques proposed in Ref. [36, 37] can treat LOTOS specifications including only action prefix and choice operators without data parameters. Ref. [19] has provided a derivation technique to deal with LOTOS specifications with data parameters and all the operators except the disabling and synchronization operators. For describing general distributed systems, it is needed to use both the data parameters and all basic operators containing the disabling and synchronization operators. However, there were no techniques for such a class.

Generally, in designing phases, the system specifications are frequently modified for debugging and/or improving the systems as the result of the analysis of system behavior. For the efficient behavior analysis, the simulation to execute the specification interactively is useful. For this purpose, several LOTOS simulators have been proposed [16, 20, 48, 52]. Most of these simulators displays the system behavior only by text information. However, since most LOTOS specifications includes a lot of operators such as choice, parallel, synchronization and so on, there are several executable events at each point of time. LOTOS specifications are also structured by packing some meaning set of events as a process, so only the textual information is inadequate to display the behavior of such specifications. It may be useful to display the structure of LOTOS specification visually and to represent the dynamic behavior of the specification by changing the visual figure corresponding to the structure step by step as each event execution. For analysis and improvement of distributed systems, it is needed to get the information about how and when communication events are executed at each node, by executing the tuple of protocol entity specifications in a distributed environment. It is also desirable to display such information visually. There were no simulators for these purposes.

Final specifications of the system should be converted into efficient object codes for a given target machine. As mentioned above, LOTOS specifications contain a number of hierarchical operators such as choice, parallel, disabling and synchronization to specify the temporal ordering of events. The implementation of the multi-rendezvous mechanism of LOTOS which enables the dynamic synchronization and the data exchange among the concurrent processes, requires

a number of communications to select a set of the appropriate processes of all running processes dynamically. So, it is complicated to implement such LOTOS specifications in the procedural languages such as C manually. For this purpose, a lot of automatic implementation techniques for LOTOS specifications have been studied [8, 13, 17, 18, 40, 41, 47]. In Ref. [17], a technique to generate sequentially executable object codes by eliminating parallelism in LOTOS specifications is presented. Ref. [13] has proposed a technique to convert a LOTOS specification into a set of parallel automata and to map each automaton to a thread in the multi-thread mechanism[1], by restricting the structure of LOTOS specifications and the available operators. However, there are no techniques to deal with an enough class to describe general distributed systems and to generate efficient object codes.

In this thesis, the following three research topics are studied as a solution for the above purposes.

(1) For a given service specification with data parameters and all basic operators in LOTOS, and an assignment of each gate[2] to a node, a technique to derive correct protocol entity specifications is proposed.

(2) A graphical LOTOS simulator which can execute a tuple of protocol entity specifications and display the dynamic behavior visually, is proposed.

(3) An implementation method for a wide class of LOTOS specifications using a multi-thread mechanism and a compiler based on the method are proposed. A real-time visualization method for LOTOS specifications using the compiler is also provided.

Above research topics are important to design and develop reliable distributed systems efficiently.

In chapter 3, a derivation algorithm for LOTOS specifications with data parameters and all basic operators such as choice, parallel, synchronization and disabling is provided. Although the proposed derivation algorithm still imposes some restrictions such that the alternative events must belong to the same node as well as the existing algorithms, there were no algorithms dealing with such a wide class. Let us suppose that each input/output event is executed at a gate, and that an assignment of each gate to a node is given by the designer. Here, we assume that each communication message is exchanged asynchronously through a reliable communication channel such as a FIFO queue between any two nodes (synchronous communication among the nodes using the synchronization operator is not considered). The derivation algorithm inputs a service specification and an assignment between gates and nodes, and outputs a tuple of protocol entity specifications. The basic steps for deriving the correct protocol entity specifications are (i) to extract the events executed in each node, and (ii) to

[1]The mechanism which can handle concurrent light weight processes efficiently in a user process of the target operating system.

[2]Here, a gate represents an interaction point to the external environment.

insert appropriate communication events between the extracted events. In order to preserve the temporal ordering of events among distributed nodes, some communication messages must be exchanged between the node executing an event and the other nodes executing the next events. When a new data is input in a node, the data may have to be transferred to other nodes which use the data. As a trivial solution, there is a way that each node broadcasts the information about each event execution with the input data, to all the other nodes. However, since the number of exchanged messages becomes large and it takes much time to execute such communication events, it is desirable to reduce the number of the communication events, especially in the real-time systems. In the derivation algorithm, the source and destination of the communication events are calculated from the relationships about the temporal ordering of events and the data dependency only when those communication events are necessary. Several functions are defined for the syntax tree of the behavior expression in the given service specification to calculate the sets of nodes which can execute first and last executable events, respectively, and the set of the pairs of the data name and node using the data. To describe the derivation algorithm easily, the derivation algorithm containing the calculation of such sets is described in an attribute grammar. The automatic derivation system is developed as a system which only evaluates the attributes.

For example, the derivation algorithm is applied to a software process consisting of several engineers. Such a software process can be considered as a distributed system and it can be described as a service specification in LOTOS. Then, each derived protocol entity specification correspond to each engineer's process description. From the service specification of the ISPW6 problem [35], an open problem for modeling software processes, the protocol entity specifications were derived in about 80 seconds using the derivation system.

Chapter 4 presents a graphical simulator which executes each protocol entity specification and displays its dynamic behavior visually. In order to represent the dynamic behavior of LOTOS specifications with its structural information such as execution dependency between several processes, it is useful to execute the specification and to display the syntax tree of the behavior expression visually. The visual representation of the syntax tree facilitates to observe both the currently executable events and the temporal ordering of events in the future from the current behavior expression. The proposed LOTOS simulator displays such a syntax tree visually, and updates the tree step by step at every event execution. In order to display the dynamic behavior of the specifications effectively, the mechanism for displaying the tree structures appropriately and rapidly. So, a fast layout algorithm provided in Ref. [39] is used to depict such tree structures. In order to facilitate to understand the structure of a large specification, the simulator can also enlarge a part of the syntax tree.

For analysis of the distributed systems, we would like to observe how and when the communication messages are exchanged among the nodes. The simulator executes the communication events contained in a protocol entity specification and actually communicates with other nodes via the network. N simu-

lators execute n protocol entity specifications by one-to-one correspondence in a distributed environment so that the whole behavior of the distributed system specified in the service specification is implemented. In order to analyze how each protocol entity specification cooperates with others in executing the events, it is desirable to understand what event is executed in the service specification while the tuple of the protocol entity specifications are executed. So, the simulator can display the current behavior expressions of both the protocol entity specification and the service specification on two graphical windows. We can also observe the behavior of n protocol entity specifications and the corresponding behavior of the service specification on $n+1$ windows on the same computer display. This facility enables the designers to develop the correct protocol entity specifications by trial and error from a service specification described in the class which the derivation algorithm cannot treat.

The simulator is also applied to enact a software process. Using the derivation algorithm, the protocol entity specifications corresponding to engineer's process descriptions can be derived from the service specification of a software process. By executing each engineer's process description at his/her workstation using the simulator, the whole software process can be enacted and the communication between engineers can be automatically exchanged. In addition, the simulator is extended to support each engineer's work. The main appended facilities are to display the currently executable events (activities of each engineer) on a menu window, and to invoke the required tools for the activity selected on the menu.

At the final stage in developing the distributed systems, the final system specifications must be implemented as efficient object codes for target machines. There have been proposed efficient implementation methods by restricting the class of the LOTOS specifications [13, 17]. For an efficient implementation using a multi-thread mechanism, Ref. [13] restricts each alternatively executed process be an action-prefix sequence (neither alternative, parallel, synchronization nor disabling composition of several subprocesses) so that the tuple of synchronizing processes is statically decided. However, we would not like to give such restrictions to LOTOS specifications. General LOTOS specifications contains the choices, interruptions and synchronizations between processes where each process also consists of several parallel and/or alternative subprocesses. In such a class, a tuple of synchronizing processes are dynamically decided depending on the selected processes. In general, if we implement such a class, a number of the communications may have to be exchanged dynamically among the processes for preserving the temporal ordering of events. So, it is important how efficiently we can implement such dynamic communications.

Chapter 5 introduces an implementation method for such a class of LOTOS specifications (including all basic operators such as choice, parallel, synchronization and disabling, and data parameters defined in abstract data types, and puts no structural restrictions in behavior expressions) within a node (processor). The proposed method uses an efficient dynamic message exchange mechanism among processes to implement LOTOS specifications where the num-

ber and the combination of synchronizing processes can be only dynamically decided. To handle concurrent processes efficiently, the method also uses a multi-thread mechanism. In the proposed method, each sequentially executable sub-expression (called *run-time unit*) is mapped to a thread in the object code with a multi-thread mechanism, and all threads are created and executed concurrently as autonomous processes in the object codes. The threads access the same shared data area (called *the control area*) so that they execute events in the order specified in the specification. A compiler which generates such multi-threaded object codes is also developed.

In the proposed method, to implement the hierarchically specified operators of LOTOS behavior expressions, a control area consists of the structured data areas where each area is used to implement the corresponding operator. The temporal ordering of events between two threads specified with choice, disabling, synchronization and so on, is implemented by accessing the corresponding area in the control area. In the object code, when each autonomous thread accesses the control area hierarchically and knows the thread can execute its events, the thread writes its intention to the corresponding areas and executes the events. The list of areas in the control area which each thread accesses step by step is called *analysis path*. The mutual exclusion mechanism in the multi-thread mechanism is used for the concurrent threads to avoid accessing the same area inadequately. To shorten the time to access the control area may contribute the efficiency in executing the derived object codes. For this purpose, several efforts are made to reduce each analysis path in the control area. The asynchronous parallel operator and choice operator among action-prefixed sequences are implemented with no areas so that the size of the control area is statically reduced in compilation. To keep the size of control area appropriately, the unnecessary parts of the area are removed and a new area is added dynamically depending on each process invocation, process termination and so on.

LOTOS specifications include the abstract data type definitions (ADTs) described in an algebraic specification language ACT ONE [14]. The execution of such ADTs in the complete class of ACT ONE requires conditional term rewriting systems, which are difficult to implement unless the class is restricted. Although there are requirements for automatic implementation of the ADTs in LOTOS specifications, most existing implementation techniques force us to implement the data type definitions manually in other compilable languages such as C.

In the proposed implementation technique, for a subset of ACT ONE, the ADTs can be converted into efficient object codes using the existing compiler for a functional language ASL/F [22, 30].

Using the above implementation method, the wider class of LOTOS specifications can be converted into the object codes which run with the multi-thread mechanism on one processor. Some experimental results have shown the compiler can generate more efficient object codes than other existing LOTOS compilers with respect to the parallel and choice execution of events.

In order to evaluate the efficiency of the generated object codes, the fa-

cility to deal with animations is added to the compiler. In general, in order to understand and monitor the dynamic behavior of the real-time systems like communication protocols, it is desirable to display the dynamic behavior of the systems visually using animations. In such visualization, we would like to describe the scenarios which contain contents of the visualization, independently and without modifying the original specifications. For this purpose, a visualization technique is proposed. In the visualization technique, using the multi-rendezvous mechanism of LOTOS, we combine a system specification and its visualization scenario with synchronization operators concerning with the events to be visualized. A tuple of the original specification and its visualization scenario is converted into the multi-threaded object code by the compiler. To describe animations, some animation primitives are introduced to be described in LOTOS. The generated object codes display the animations using the animation server mechanism proposed in Ref. [50].

When we tried to visualize Dijkstra's "five dining philosophers" based on the visualization technique, the overhead time taken to synchronize events between the original specification and its visualization scenario was about 20% of the whole execution time of the original specification. The experimental results have shown that the generated object codes can display animations in real-time.

In this thesis, for a wide class of LOTOS, the techniques deriving protocol entity specifications from a service specification, executing and visualizing a tuple of protocol entity specifications with a simulator, and implementing each protocol entity specification as an efficient object code are presented. The results of the above researches may contribute the efficient design and development of the reliable distributed systems.

Hereafter, in Chapter 2, the brief explanation of the formal specification language LOTOS is described, and the service specification in LOTOS and the corresponding protocol entity specifications are explained. In Chapter 3, a derivation technique of protocol entity specifications from a service specification is introduced. In Chapter 4, a graphical LOTOS simulator which executes each derived protocol entity specification and display its dynamic behavior visually is provided. And the application of the simulator is also presented. In Chapter 5, a technique to implement LOTOS specifications using the multi-thread mechanism is explained, and the compiler based on the technique is provided. A visualization technique of LOTOS specifications which displays the dynamic behavior of the specifications using the compiler, is also proposed.

Chapter 2

How to Describe Distributed Systems in LOTOS

2.1. A formal specification language LOTOS

2.1.1 Behavior expressions

In LOTOS, we describe a specification as a process which consists of several sub-processes. In each process, we specify the events observed from the external environment and their temporal ordering as a behaviour expression. The operators shown in Table 2.1 can be used to specify the sequential execution of events, alternative, synchronized and parallel execution, interruption and so on. In Table 2.1, α represents an event, and B, B_1 and B_2 represent expressions consisting of some events, some process invocations and the above operators. In (4), $g_1, ..., g_k$ are gate names whose events must be synchronized between B_1 and B_2 ('||' also means that all events of B_1 and B_2 must be synchronized each other). In (7), an invocation of a sub-process P is specified.

2.1.2 Abstract data types

In LOTOS, the algebraic specification language ACT/ONE [14] is used for defining the data types and their operations which are used to describe guard expressions and/or I/O values in each event [31].

Table 2.1: LOTOS operators

(1) **action-prefix**	$\alpha; B$
(2) **choice**	$B_1 [] B_2$
(3) **parallel**	$B_1 \vert\vert\vert B_2$
(4) **synchronous parallel**	$B_1 \vert[g_1, \cdots, g_k]\vert B_2$, $B_1 \vert\vert B_2$
(5) **disabling**	$B_1 [\rangle B_2$
(6) **enabling**	$B_1 \gg B_2$
(7) **process instantiation**	$P[g_1, \cdots, g_n](v_1, \cdots, v_m)$

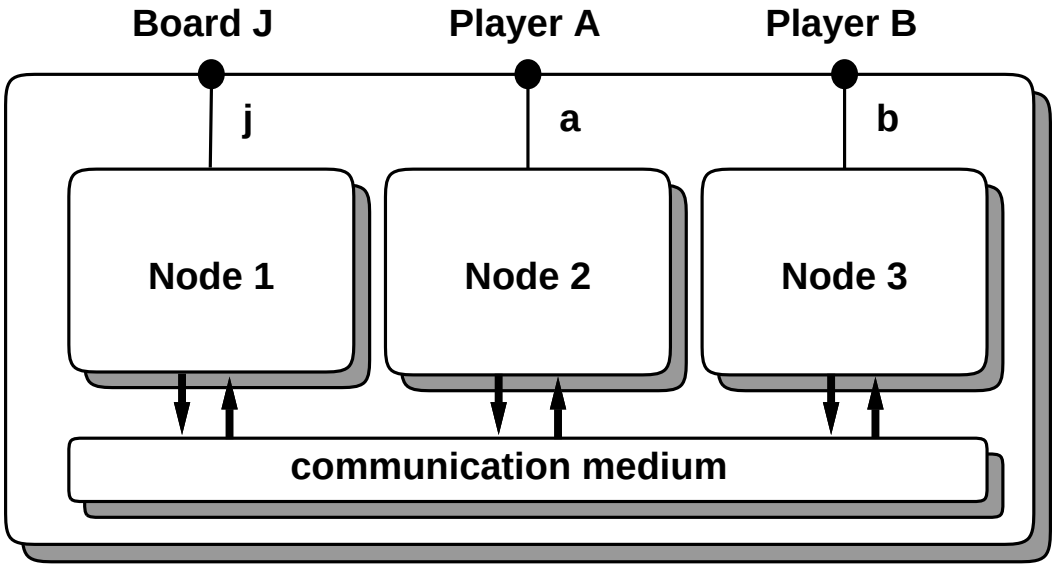


Figure 2.1: Janken game in a distributed system

2.2. Service specification and its correct protocol specification

In this section, we explain the difference between a service specification and the corresponding protocol specification definitely. Then, we give the formal definition of the correctness of the protocol specification.

2.2.1 Service specification

Even if several nodes cooperate to provide a service, we do not describe the exchange of the synchronization messages and data values in a service specification.

For explanation, we will use *Janken Game*. *Janken Game* is a finger-flashing game of *paper-scissors-stone*. *Paper* wins against *stone*. *Stone* wins against *scissors*. *Scissors* wins against *paper*. Each player chooses one of them. For instance, if the players A and B choose *stone* and *scissors*, respectively, then the player A wins the game. If they choose the same one, then they try the game again. Suppose that we play Janken Game in a distributed system (Fig. 2.1).

There are a Board J and two players A and B. In Fig. 2.1, we assume that there are three nodes 1, 2 and 3. These nodes correspond to the Board J and two players A and B, respectively. They have the gates "j", "a" and "b", respectively.

First, the node 1 shows the "start" of the game to the Board J(gate "j"). Then two players give their choices at the gates "a" and "b", respectively. The system decides the winner and shows it to the Board J. If the game is a tie, then

Table 2.2: Service specification of Janken Game

```
specification Janken[j,a,b]:exit
(* the description of the data types used in this program *)
behavior  Game[j,a,b]
where
  process Game[j,a,b]:=
    j!"start" ; ( a?x:finger ; exit ||| b?y:finger ; exit )
>> (    ( [result(x,y)="A"] -> j!result(x,y) ;
                        (a!"win" ; exit ||| b!"loss"; exit ) )
     [] ( [result(x,y)="B"] -> j!result(x,y) ;
                        (a!"loss"; exit ||| b!"win" ; exit ) )
     [] ( [result(x,y)="tie"] -> j!result(x,y); Game[j,a,b] )
   )
  endproc
endspec
```

the string "tie" is shown to the Board J and the game is carried out again. If either two players A or B wins the game, then the strings "win" and "loss" are shown to the gates of the winner and loser, respectively. In Table 2.2, a service specification of this game is described in LOTOS.

In Table 2.2, "result(x,y)" is a function which calculates the winner of this game. It returns the string "A" (or "B") if "x" wins (loses) against "y". Otherwise, it returns the string "tie".

2.2.2 Protocol specification

On the level of the protocol specification, the communication medium is considered explicitly. If some nodes must cooperate to provide a service, they must synchronize each other in order to keep the temporal ordering of the execution of the events.

LOTOS has rendezvous communication mechanisms among more than two nodes. However, we treat only asynchronous communication between two nodes. For this restricted class, some techniques synthesizing protocol specifications have been proposed [19, 24, 34, 36]. Also many practical protocols can be described in this class [34]. Therefore, we think the class is still useful. We have easily implemented the communication mechanisms of this class using inter process communication(IPC) of UNIX system. Hereafter, we assume asynchronous communication.

For example, in order to execute an event "a" at a node "i" and then to execute an event "b" at another node "j", the node "i" must send a synchronization message to the node "j" after "a" is executed. The node "j" must execute "b" after the node "j" receives the synchronization message. If a node uses the data values which it does not know, the node must receive the values from the

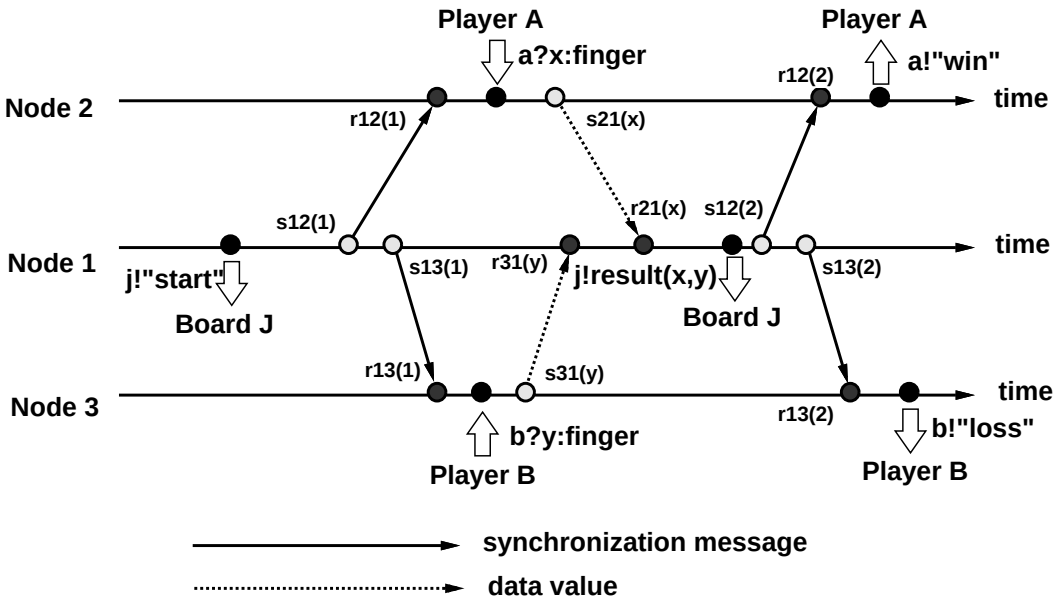


Figure 2.2: An execution process of protocol specification

nodes which know them. For example, if a value of a variable "x" is inputted at a node "i" and another node "j" needs the value, then the value must be transmitted from the node "i" to the node "j". So, we must determine what kinds of the synchronization messages and data values should be sent and/or received in each node. We must also determine the timing of the sending/receiving actions and the destinations of the sending/receiving messages and data.

We assume that there is a reliable asynchronous communication channel from each node "i" to any other node "j". That is, we assume that each message sent from the node "i" is eventually received by the node "j". Each channel does not lose, duplicate, nor insert messages. At each node "i", the sending action of a synchronization message "m" to a node"j" is described as the special event "$\mathrm{s}_{ij}(m)$". And the receiving action of a message "m" from a node "j" is described as "$\mathrm{r}_{ji}(m)$". There are two types of messages : synchronization messages and data values. We use the integers as the varieties of the synchronization messages. Let "k" be an integer and let "x" be an variable. The events $\mathrm{s}_{ij}(k)$ and $s_{ij}(x)$ represent the transmission of the synchronization message "k" and the data value of the variable "x", respectively.

In Table 2.3, we will give a protocol specification which provides the service described in Table 2.2. An execution process of the protocol specification in Table 2.3 is illustrated in Fig. 2.2.

First, the node 1 executes the event j!"start" and sends the synchronization messages "1" to the nodes 2 and 3. If the nodes 2 and 3 receive the messages, then they read the choices of the players A and B and inform them to the node 1. Then, the node 1 decides the winner and shows it to the Board. If the winner

Table 2.3: Protocol specification of Janken Game

(1) Node 1

```
specification Janken_1[j,s12,s13,r21,r31]:exit
... (the description of the data types used in this program)
behavior  Game_1[j,s12,s13,r21,r31]
where
process Game_1[j,s12,s13,r21,r31] :=
     j!"start" ; ( s12(1) ; exit ||| s13(1) ; exit  )
      >> (r21(x);exit ||| r31(y);exit)
      >> (  ([result(x,y)="A"] -> j!result(x,y) ;
                           (s12(2);exit ||| s13(2);exit )  )
         [] ([result(x,y)="B"] -> j!result(x,y) ;
                           (s12(3);exit ||| s13(3);exit )  )
         [] ([result(x,y)="tie"] -> j!result(x,y) ;
                           (s12(4);exit ||| s13(4);exit)
        >> Game_1[j,s12,s13,r21,r31]   )
         )
  endproc
endspec
```

(2) Node 2

```
specification Janken_2[a,r12,s21]:exit
... (the description of the data types used in this program)
behavior  Game_2[a,r12,s21]
where
  process Game_2[a,r12,s21]:=
    r12(1) ; a?x:finger ; s21(x) ;
        (   ( r12(2) ; a!"win" ; exit    )
         [] ( r12(3) ; a!"loss"; exit    )
         [] ( r12(4) ; Game_2[a,r12,s21] )
        )
  endproc
endspec
```

(3) Node 3

```
specification Janken_3[b,r13,s31]:exit
... (the description of the data types used in this program)
behavior  Game_3[b,r13,s31]
where
  process Game_3[b,r13,s31]:=
   r13(1) ; a?y:finger ; s31(y) ;
        (   ( r13(2) ; b!"loss"; exit    )
         [] ( r13(3) ; b!"win" ; exit    )
         [] ( r13(4) ; Game_3[b,r13,s31] )
        )
  endproc
endspec
```

is A (or B), then it sends the synchronization messages "2" (or "3") to the nodes 2 and 3. The nodes 2 and 3 know the winner by receiving these messages. If they receive the synchronization message "4", then the processes Game_2 and Game_3 are invoked again.

2.2.3 Correctness of protocol specification

In this section, we will define the correctness of protocol specifications formally. Let P and Q be processes written in LOTOS. If the processes P and Q are observational equivalent [31], then we describe "P≈Q". Let P_k be a specification of the node "k". For a service specification P_S and a protocol specification $<P_1,\cdots,P_N>$ with N nodes , we say that the protocol specification is correct with respect to the service specification P_S if the following relation holds.

$P_S \approx$ hide G in (P_1 ||| P_2 ||| $\cdots$ ||| P_N) |[G]| Comm

Here,

$G = \{ s_{ij}, r_{ij} \mid 1 \leq i,j \leq N,\ i \neq j \}$

$Comm = Comm_{12}$ ||| $\cdots$ ||| $Comm_{ij}$ ||| $\cdots$ ||| $Comm_{N-1\ N}$
$(1 \leq i,j \leq N, i \neq j)$

$Comm_{ij} = s_{ij}$; r_{ij} ; $Comm_{ij}$

"Hide H in Q" represents that the events of the process Q belonging to H are treated as internal events. Internal events are not observed from the external environments. And "P |[F]| Q" denotes that two processes P and Q are executable in parallel, and that the events of P and Q belonging to F must be executed simultaneously. The expression "(P_1||| $\cdots$ ||| P_N) |[G]| Comm" requires that a receiving action should not occur prior to a sending action in each communication channel, and that if a sending action is executed, then the corresponding receiving action must be executed eventually. And "**hide** G" represents that the sending/receiving actions are treated as internal events.

For example, since the service specification "Janken" in Table 2.2 and the protocol specification <Janken_1, Janken_2, Janken_3> in Table 2.3 satisfy the following relation, the protocol specification is correct.

Janken[j,a,b] ≈ hide G in (Janken_1[j,s12,s13,r21,r31]
||| Janken_2[a,r12,s21]
||| Janken_3[b,r13,s31]) |[G]| Comm

Here, G = {s12,s13,r21,r31,s21,r12,s31,r13}

$Comm = Comm_{12}$||| $Comm_{13}$||| $Comm_{21}$||| $Comm_{31}$

$Comm_{ij} = s_{ij}$; r_{ij} ; $Comm_{ij}$

We can show that the above protocol specification is correct by using the technique in Ref. [49].

Chapter 3

Derivation of Protocol Specifications from Service Specifications in LOTOS with Data Parameters

3.1. Introduction

In this chapter, a derivation algorithm for LOTOS specifications with data parameters and all basic operators such as choice, parallel, synchronization and disabling is provided.

The derivation algorithm inputs a service specification and an assignment between gates and nodes, and outputs a tuple of protocol entity specifications. The basic steps for deriving the correct protocol entity specifications are (i) to extract the events executed in each node, and (ii) to insert appropriate communication events between the extracted events. In order to preserve the temporal ordering of events among distributed nodes, some communication messages must be exchanged between the node executing an event and the other nodes executing the next events. When a new data is input in a node, the data may have to be transferred to other nodes which use the data.

In the derivation algorithm, the source and destination of the communication events are calculated from the relationships about the temporal ordering of events and the data dependency only when those communication events are necessary. Several functions are defined for the syntax tree of the behavior expression in the given service specification to calculate the sets of nodes which can execute first and last executable events, respectively, and the set of the pairs of the data name and node using the data. To describe the derivation algorithm easily, the derivation algorithm containing the calculation of such sets is described in an attribute grammar. The automatic derivation system is developed as a system which only evaluates the attributes.

In this chapter, the derivation algorithm is explained using a software process consisting several engineers as an example of a service specification. The derivation system is also developed to evaluate the usefulness of the proposed

algorithm.

3.2. Developing distributed systems using LOTOS

In this chapter, as an effective example of distributed systems, we give a service specification of a software process consisting of several engineers in LOTOS.

3.2.1 Definition

First, we give notations to describe software processes in LOTOS. We assume that each engineer's activity in a software process is either an input action, an output action or an input/output action (where several data are input and output). In LOTOS, an activity is described as an event as follows:

$$Person!x?y : type \cdots [x = tool1(x), y = tool2(z), \cdots]$$

Here, we call $Person$ as an *engineer's ID*, or simply an *engineer*, which corresponds to a member in a process. we call the expression $!x?y : type \cdots$ as an *input/output part*, in which an input action($?y : type$), an output action($!x$) or an input/output action($!x?y : type \ldots$) is described. We also call the part nipped in [] as a *work content*. In a work content, we describe some tool names activated when an activity is executed. An output action to an engineer such as preview of a file is described as $!x[x = tool1(x)]$, where x is a variable(data) representing a file name, and $x = tool1(x)$ represents a data x is output using a tool $tool1$. Similarly, an input action to a computer such as editing of a file is described as $?y : type[y = tool2(z)]$, where y is a data of type $type$, and $y = tool2(z)$ represents that a data y is input to a computer using a tool $tool2$ with an existing data z. An input/output action is the combination of the above two. A new data is generated only in an input or input/output action. In LOTOS, conditional expressions can be also described in a work content [31]. For example, the activity "SE$?w : int[w > 0]$" cannot be executed until the integer value more than 0 is input to the variable w.

We can omit both an input/output part and a work content in an activity. If neither part is described, we call the activity as a *dummy activity* (which performs nothing). If only an input/output part is described, data is input(output) to(from) a computer via standard input(output).

For example, an activity such that a system engineer SE modifies a system specification(fully, he edits an old file of the specification *oldspec*, and makes a new file *spec*) is described as follows in LOTOS:

$$\text{SE?spec} : \text{file[spec} = \text{edit(oldspec)]}$$

Here, we suppose that *edit* means a tool for editing files(such as *emacs* or *vi*). A file generated by above activity is assigned to a variable *spec*.

3.2.2 Describing service specification

We describe a service specification by specifying activities of all engineers in a process and their temporal order. The operators of LOTOS in Table 2.1 are used to specify the order. For example, a sequence of successive activities such that 1. a system engineer(SE) gives a specification of a system(*spec*), and then 2. a programmer(PG) makes a program (*code*) based on the specification(*spec*), is described as follows:

```
SE?spec:file[spec=edit()];
PG!spec?code:file[spec=view(spec),
                  code=makeprog()];  exit
```

(Here, we suppose that *edit*, *view* and *makeprog* are tools for editing files, displaying files and making programs, respectively.) The operator ";" connects two activities, and offers successive execution of them. Any number of activities can be connected using ";" for successive execution. An operator *exit* is placed at the end of the sequence of successive activities. A process identifier may be also placed at the end of the sequence instead of *exit*(this means process invocation). We call such a sequence simply as a *successive sequence*. In LOTOS, selective execution, asynchronized parallel execution, synchronized parallel execution, successive execution and interruption between any two successive sequences can be described using the operators in Table 2.1.

Now, we try to describe the following simple software development process *MakeCode* in LOTOS as an example.

All activities and their temporal order in the process MakeCode

1. SE edits an old specification of the system *oldsp*, and makes a new specification *spec*.

2. P1, P2 and QE edit old versions of two source programs and a test data, *olds*1, *olds*2 and *oldtd*, and make new versions *src*1, *src*2 and *tstdt*, respectively, in parallel.

3. QE makes an executable module of the system *code* by compiling the source programs *src*1 and *src*2 and linking their objects.

4. QE tests the executable module *code* with the test data *tstdt* and decides whether the process should be repeated or terminated(he puts "NO" for repetition and "OK" for termination into a variable *res*).

5. SE selects process repetition or process termination from the QE's testing result *res*.

6. Whenever the process is running, SE can interrupt the process.

We give the service specification of MakeCode in Table 3.1.

```
process   MakeCode[SE,P1,P2,QE]
          (oldsp:file, olds1:file, olds2:file, oldtd:file):exit :=
 (( SE?spec:file[spec=makedocument(oldsp)];
    ( P1!spec?src1:file[spec=view(spec)
          ,src1=edit(olds1)]; exit
    ||| P2!spec?src2:file[spec=view(spec)
          ,src2=edit(olds2)]; exit
    ||| QE!spec?tstdt:file[spec=view(spec)
          ,tstdt=edit(oldtd)]; exit ) )
  ≫
  ( QE?code:file[code=compile(src1,src2)];
    QE?res:string[res=testcode(code,tstdt)];
          (SE!"Try again"[res="NO"];
          MakeCode[SE,P1,P2,QE](spec,src1,src2,tstdt)
          [] SE!"OK"[res="OK"]; exit ) )
 )
 [⟩ SE!"Interruption"; exit
endproc
```

Table 3.1: Service specification of "MakeCode"

3.2.3 Outline for deriving protocol specifications

Outline for deriving protocol specifications

In order to derive protocol entity specifications from a service specification, it is necessary to be able to deal with communications among engineers (nodes) in LOTOS. We describe the communication actions in LOTOS as follows:

- the action sending message m to the engineer n ... $\mathrm{s}_n(m)$
- the action receiving message m from the engineer n ... $\mathrm{r}_n(m)$

We also describe sending actions from one to several as $\mathrm{s}_{\{P1,P2,QE\}}(m_1)$, for example, and receiving actions from several to one as $\mathrm{r}_{\{SE,P1\}}(m_2)$. Here, above communication actions are treated as the activities. First, we explain how to derive protocol entity specifications from a service specification using the example in Table 3.1. For simplicity of discussion, let's consider the case that the interruption in MakeCode does not occur. Deriving steps are (1) extracting all activities of an engineer, and (2) deriving necessary communications by searching who performs the activities located before and after each extracted activity.

We focus on the programmer P1. The activity of P1 is (1) "P1!spec?src1:file [...]", and the process invocation in this process is (2) "MakeCode[SE,P1,P2,QE] (...)". In order to execute the activity (1), P1 needs the data *spec* in the previous activity "SE?spec ...". The engineer's IDs of the activities located before and after the activity (1) are SE and QE, respectively. The data *src*1 made in the activity (1) is used in the succeeding QE's activity. Consequently, in order that the process runs cooperatively, P1 must execute the activity (1) after receiving the data *spec*, and then he must send the source code *src*1 to QE. In the process invocation (2), since the engineer's ID of the activity located before

it is SE, P1 must receive the synchronization message for the process invocation from SE and invoke the process after that. P1 may also receive another message when SE decides the process termination because either the process invocation (2) or process termination is performed selectively. It is obvious that the temporal order of P1's activities should be as follows:

r_{SE}(spec); P1!spec?src1:file[...];
  s_{QE}(src1);
  (r_{SE} (message for process invocation);
      MakeCode[...](...)
  [] r_{SE}(message for process termination);
      exit)

Communications to be derived depend on the types of the LOTOS operators in a service specification. The details are described in Section 3.3.

Execution of protocol specifications

A process can be enacted by executing all protocol entity specifications in parallel. We have developed a support system for process enaction (the system is introduced in Section 4.4). Here we outline the facilities which the system offers. The main facilities are (1) automatic exchange of communication messages among engineers, and (2) effective guidance of executable activities for each engineer. The facility (1) enables the engineers not to make mistakes about communications and to concentrate on their actual works since communication messages among engineers are exchanged automatically and reliably. The facility (2) lists the contents of all executable activities at each point of time on a menu in X-window system. It guides each engineer what to do at each point of time. Since the data and development tools required in an activity are automatically selected and activated by the system, no mistakes about them will occur. The above facilities (1) and (2) enables the engineers only to perform the contents displayed on the menu successively so that the process proceeds in order of specifying in a service specification. The system can also display the current expressions of the service specification and each engineer's protocol entity specification at each point of time graphically on his display. This informs the engineer the current status of the process.

Fig.3.1 shows a snapshot of our system. Here, the protocol entity specification of P1 derived from the service specification in Table 3.1 is executed by the system. The protocol entity specifications of others are similarly executed elsewhere. Executing steps in our system are as follows:

step1 The system displays the syntax trees of the current protocol entity specification and service specification on graphic windows(*MakeCode(P1)* and *MakeCode(Whole Process)* in Fig.3.1). The dotted rectangles in the windows show executable activities(here, P1 can execute the activity *P1!spec ?src1:file [spec = view(spec), src1=edit(olds1)]*).

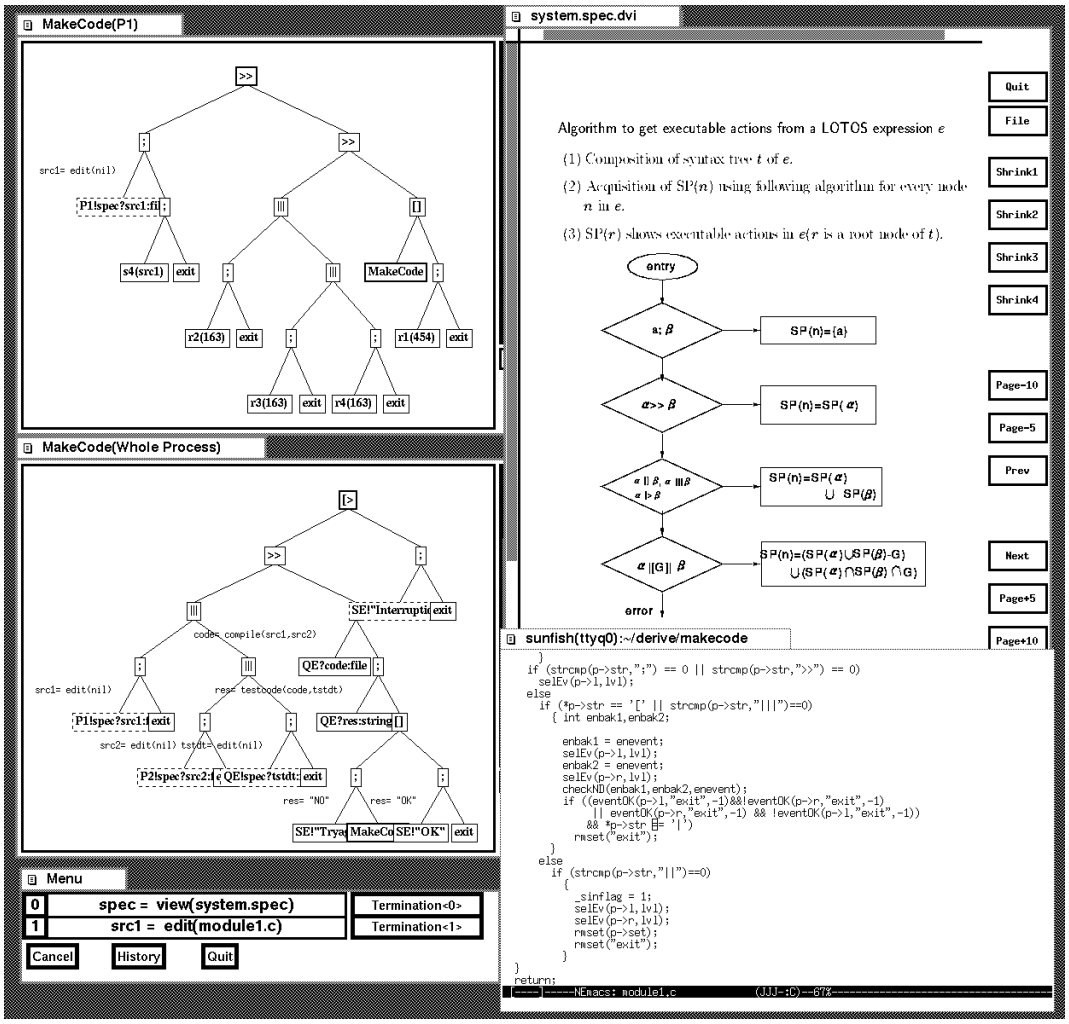


Figure 3.1: Snapshot of support system

step2 It calculates the executable activities in the current protocol entity specification and displays the contents of them as a menu(*Menu* in Fig.3.1).

step3 If the engineer selects one on the menu, then the system activates the tools required in the activity. He can terminate the work in the activity by clicking the termination button on the menu.

step4 It rewrites the current protocol entity specification and service specification into the new ones where the performed activity in the above **step 3** is removed.

step5 The **steps 1** to **4** are repeated for the new protocol entity specification and service specification.

In Fig.3.1, two items are shown on a menu, one is "spec = view(system.spec)" for displaying the file *system.spec* and the other is "src1 = edit(module1.c)" for editing of the file *module*1.*c*. Each engineer may want to use his favorite editor for editing the file. We call such selection the *customization* of tools. Our system offers the facility for the customization. Here, actual programs *xdvi* and *emacs* are activated as tools *view* and *edit*, respectively. The details are explained in Section 4.4.

3.3. Deriving protocol specifications

For deriving protocol entity specifications from a service specification, some algorithms for Basic LOTOS and its subclasses have been proposed [4, 34, 37] where they do not handle the abstract data types. We have extended the algorithm in Ref.[34] to a class which can handle the abstract data types. We introduce this extended algorithm in this section.

In this section, a derivation algorithm for LOTOS specifications with data parameters and all basic operators such as choice, parallel, synchronization and disabling is provided. Although the proposed derivation algorithm still imposes some restrictions such that the alternative events must belong to the same node as well as the existing algorithms, there were no algorithms dealing with such a wide class.

Let us suppose that each input/output event is executed at a gate, and that an assignment of each gate to a node is given by the designer. Here, we assume that each communication message is exchanged asynchronously through a reliable communication channel such as a FIFO queue between any two nodes (synchronous communication among the nodes using the synchronization operator is not considered).

3.3.1 Derivation algorithm

Basic steps deriving protocol entity specifications from a service specification are (1) to extract events of a node p from the service specification e and (2) to

insert appropriate communications of the node p before and after the extracted events.

Attribute grammar

For derivation of protocol entity specifications, first we parse the expression(event sequence) of a service specification and get its syntax tree(hereafter, we call it an *evaluation tree*). Secondly, we give the following six attributes to each node of the evaluation tree. Here, e and p represent any subtree of the evaluation tree and any distributed node (processor), respectively. For each event "$Person?x!y\cdots$", we call the pairs $\langle Person, x\rangle$ and $\langle Person, y\rangle$ as *data ID*s. Using data IDs, we can decide who use each data.

SP(e) a set of nodes performing the initial events in a subtree e

EP(e) a set of nodes performing the last events in e

AP(e) a set of nodes performing the events in e

VA(e) a set of data IDs in e

VN(e) a set of data IDs contained in all the evaluation tree

PS(e, p) p's protocol entity specification for a subtree e

SP, EP, AP, VA and PS are *synthesized attributes* [2], whose values are calculated from the leaf nodes to the root node of an evaluation tree. VN is an *inherited attribute* [2] where the attribute value of VA at the root node is passed to its all descendants as the attribute values of VN.

Calculating above attributes to the evaluation tree in the following order, protocol entity specifications can be derived.

1. Calculating the attribute values of SP, EP, AP and VA for all nodes in the evaluation tree
2. Calculating the attribute values of VN for all nodes
3. For each node p, calculating the attribute values of PS for all nodes

After above steps, the attribute value of PS at the root node r for a node p, PS(r, p) is a node p's protocol entity specification. Attributes SP, EP and AP are used to know whose events are located before and after each event of p. Attributes VA and VN are used to know what data are used in events and who use them.

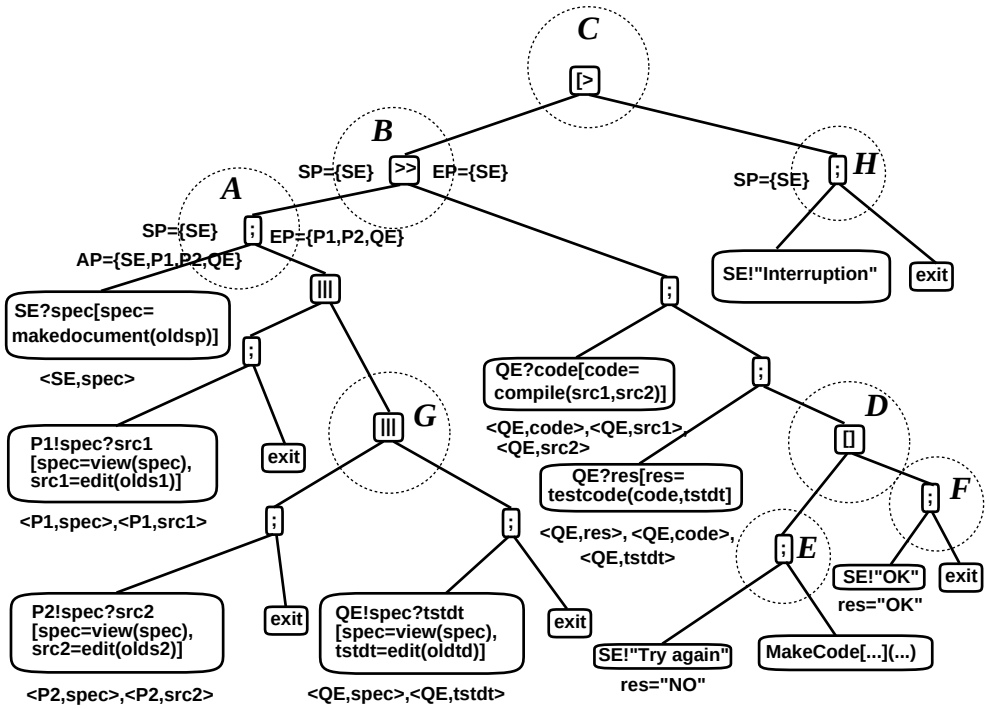


Figure 3.2: Syntax tree of the service specification of 'MakeCode'

Derivation technique for each operator

The evaluation tree(syntax tree) for the service specification in Table 3.1 is shown in Fig.3.2. The values of attributes SP, EP and AP at node A in the tree are {SE}, {P1, P2, QE}, {SE, P1, P2, QE}, respectively. Attribute values of PS are calculated using the attribute values of SP, EP, AP and VN. PS's attribute values depend on the operator of each node. In the below discussion, the term "node" represents an engineer in the software processes.

• **Successive sequence** $a;\beta$

$a;\beta$ means that the event sequence β gets executable after the event a is executed.

(i) In the case that a **makes no new data**

In this case, the node of the event a sends synchronization messages to the node of SP(β) after the event a is executed. The node of SP(β) receive the messages and then execute the events in β.

(ii) In the case that a **makes a new data** x

If a new data x is generated in the event a, communication messages are added to the protocol entity specifications in following order.

1. The node of the event a executes it.

2. He sends the data x to the nodes $\mathrm{V} = \{n | \langle n, x\rangle \in \mathrm{VN}(a)\}$ using the data x in their events and the nodes V receive x respectively.

3. The nodes V send synchronization messages to the nodes of SP(β) and the nodes of SP(β) receive the messages.

4. The nodes of SP(β) execute events in β.

(iii) In the case that β is the process ID($a; P$)

In the case that β is the process identifier P, since the process P is invoked after a has been executed, the synchronization messages should be sent from the node of a to AP(P) after a is executed.

As an example, we calculate the values of PS at node A in Fig.3.2. Since the initial event "SE?spec" makes a new data *spec*, sending actions of data *spec* from SE to $\{n|\langle n, spec\rangle \in \mathrm{VN(SE?spec)}, n \neq \mathrm{SE}\} = \{P1, P2, QE\}$ are added to the protocol entity specification of SE and corresponding receiving actions are also added to the protocol entity specifications of other nodes. Let a right subtree under the node A be β(whose root node is |||). P1, P2 and QE send synchronization messages for the data receptions to the nodes of SP(β)={P1, P2, QE}(here, messages from the node to itself are omitted). Above steps produce the following partial protocol entity specifications. Here, the node label A is used as a synchronization message.

service specification
SE?spec:file[...];
(P1!spec?src1:file[...]; exit
||| P2!spec?src2:file[...]; exit
||| QE!spec?tstdt:file[...]; exit) ...

protocol entity specifications
SE: (SE?spec:file[...]; $\mathrm{s}_{\{P1,P2,QE\}}$(spec); exit)
$\gg$ ...
P1: ($\mathrm{r}_{\{SE\}}$(spec) ; exit)
$\gg$ ($\mathrm{s}_{\{P2,QE\}}$(A); exit)
$\gg$ ($\mathrm{r}_{\{P2,QE\}}$(A); exit)
$\gg$ (P1!spec?src1:file[...]; ...
(protocol entity specifications of P2 and QE are similar to P1's)

• **Selective execution $\alpha[]\beta$**

From a simple service specification "SE?x; P1!x; exit [] SE?y; exit", if we apply the algorithm for successive sequences to both sides of the operator [] in the expression independently, the following protocol entity specifications are derived.

SE: SE?x; s_{P1}(x); exit [] SE?y; exit
P1: r_{SE}(x); P1!x; exit [] exit

In this case, if SE selects the left side expression of [], and P1 selects the right side expression *exit* before receiving data x from SE, consistency of the service

specification is not guaranteed in the protocol entity specifications. Therefore synchronization messages must be exchanged between the nodes executing initial events in each side of [] and the nodes belonging not to one sequence but to another sequence.

For simplicity of the algorithm, we assume that SP(α)=SP(β)=$\{p'\}$. This means that each node executing initial events in α or β is merely a node p'(the derivation technique in the case that SP(α) $\neq$ SP(β) is introduced in Ref.[37]). We also assume that VA(α)=VA(β) for preserving consistency of data. In these assumptions, the protocol entity specification of p', PS($\alpha[]\beta, p'$) is derived as follows:

$$
\begin{aligned}
&(\text{PS } (\alpha, p') \gg \\
&\quad \text{s}_{AP(\beta)-AP(\alpha)}(\text{synchronization message}) \ ) \\
&[] \ (\text{PS}(\beta, p') \gg \\
&\quad \text{s}_{AP(\alpha)-AP(\beta)}(\text{synchronization message}) \ )
\end{aligned}
$$

Similarly, the protocol entity specification of other node p, PS($\alpha[]\beta, p$) is derived as follows:

$$(\text{PS}(\alpha, p) \gg \gamma(\alpha, \beta)) \ [] \ (\text{PS}(\beta, p) \gg \gamma(\beta, \alpha))$$

Here,

$$
\begin{aligned}
\gamma(\alpha, \beta) = \mathit{if} \ & p \in \text{AP}(\beta) - \text{AP}(\alpha) \\
& \mathit{then} \ \ \text{r}_{p'}(\text{synchronization message}) \\
& \mathit{else} \ \ \text{exit}
\end{aligned}
$$

At node D in Fig.3.2, for example, the protocol entity specifications are derived as follows using the node labels E and F of two subtrees under [].

SE: (((SE!"Try again"[res="NO"]; exit)
  $\gg$ (s$_{\{P1,P2,QE\}}(E)$; exit)
  $\gg$ MakeCode ...)
  [] ((SE!"OK"[res="OK"]; exit)
  $\gg$ (s$_{\{P1,P2,QE\}}(F)$; exit)))

P1: (((r$_{SE}(E)$; exit) $\gg$ MakeCode ...)
  [] (r$_{SE}(F)$; exit))

(protocol entity specifications of P2 and QE are similar to P1's)

• **Successive execution** $\alpha \gg \beta$

In this case, after execution of events in α the nodes of EP(α) send synchronization messages to the nodes of SP(β).

• **Interruption** $\alpha[\rangle\beta$

Here, we assume that EP(α)=SP(β)=$\{p'\}$. If all events in α are executed without interruption, the node of the last event in α sends messages informing no interruption to all nodes in α and β except himself and the nodes receiving the

messages recognize no interruption. If interruption by the initial event in β occurs during the execution of the events in α, the node of the initial event in β sends messages informing interruption to all nodes in α and β except himself. The nodes receiving the messages recognize the occurrence of interruption.

Interruption may occur without an instruction of the node p' if sending actions are executed automatically by the system. We cope with this problem inserting dummy event of the node p' of initial event in β. In order to interrupt α by β, we assume that the node p' executes the dummy event $\underline{p'}$. For the node p', $\mathrm{PS}(\alpha[\rangle\beta, p')$ is derived as follows:

$$(\mathrm{PS}\ (\alpha, p') \gg \mathrm{s}_{AP(\alpha[\rangle\beta)-\{p'\}}(\text{synchronization message 1})$$
$$[\rangle(\underline{p'}; \mathrm{s}_{AP(\alpha[\rangle\beta)-\{p'\}}(\text{synchronization message 2}) \gg \mathrm{PS}(\beta, p'))$$

Similarly, the protocol entity specification of other node except p', $\mathrm{PS}(\alpha[\rangle\beta, p)$ is derived as follows:

$$(\mathrm{PS}(\alpha, p) \gg \gamma(1))\ [\rangle\ (\gamma(2) \gg \mathrm{PS}(\beta, p))$$

Here,

$$\gamma(i) = if\ \ p \in \mathrm{AP}(\alpha[\rangle\beta) - \{p\}\ \ then\ \ \mathrm{r}_{p'}(\text{synchronization message } i)$$
$$else\ \ \text{exit}$$

At node C in Fig. 3.2, since $\mathrm{EP}(\alpha) = \mathrm{SP}(\beta) = \{\ \mathrm{SE}\ \}$ in $\alpha[\rangle\beta$, above synchronization messages are send from SE to P1, P2 and QE. Using the node labels B and H in Fig. 3.2, protocol entity specifications of this part are described as follows:

SE: $(\mathrm{PS}(\alpha,\ \mathrm{SE}) \gg (\mathrm{s}_{\{P1,P2,QE\}}(B);\ \mathrm{exit})\ )$
$[\rangle\ ((\mathrm{SE};\ (\mathrm{s}_{\{P1,P2,QE\}}(H);\ \mathrm{exit})\ ) \gg \mathrm{PS}(\beta,\ \mathrm{SE})\ )$
P1: $(\ \mathrm{PS}(\alpha,\ \mathrm{P1}) \gg (\mathrm{r}_{SE}(B);\ \mathrm{exit})\ )$
$[\rangle\ (\ (\mathrm{r}_{SE}(H);\mathrm{exit}) \gg \mathrm{PS}(\beta,\ \mathrm{P1})\ )$
(protocol entity specifications of P2 and QE are similar to P1's.)

- **Parallel composition $\alpha|||\beta$ and $\alpha|[\mathrm{Q}]|\beta$**
  $\mathrm{PS}(\alpha|||\beta, p) = \mathrm{PS}(\alpha, p)|||\mathrm{PS}(\beta, p)$
  $\mathrm{PS}(\alpha|[\mathrm{Q}]|\beta, p) = \mathrm{PS}(\alpha, p)|[\mathrm{Q}]|\mathrm{PS}(\beta, p)$

Here, we restrict that neither of events in α and β assign the value to the same variable x.

In combination of above techniques for all operators, we can get the full protocol entity specifications. We show the protocol entity specifications of SE, P1, P2 and QE in Table3.2.

In our algorithm, we suppose that communication messages are exchanged through reliable FIFO channels in which messages are never lost. The algorithm requires the initial events in the both sequence of selective execution to belong to the same node. We can cope with this restriction by inserting appropriate dummy events into the service specification.

Table 3.2: Protocol specification of 'MakeCode'

Protocol entity specification of SE

```
process MakeCode[SE](oldsp: file):exit :=
 ( (SE?spec:file[spec=makedocument(oldsp)];exit)
   ≫ (s_{P1,P2,QE}(spec);exit) )
≫ ( (r_QE(res);exit)
 ≫ ( ( ( SE!"Try Again"[res="NO"]; exit )
          ≫ (s_{P1,P2,QE}(E); exit)
          ≫ MakeCode[SE](spec) )
     [] ( ( SE!"OK"[res="OK"]; exit )
          ≫ (s_{P1,P2,QE}(F); exit) ) ) )
 ≫ (s_{P1,P2,QE}(B); exit) )
[⟩   ( ( SE; s_{P1,P2,QE}(H); exit)
       ≫ (SE!"Interruption"; exit) )
endproc
```

Protocol entity specification of QE

```
process MakeCode[QE](oldtd: file):exit :=
 ( ( (r_SE(spec); exit)
 ≫ (s_{P1,P2}(A); exit) ≫ (r_{P1,P2}(A); exit)
 ≫ (r_P1(src1); exit ||| r_P2(src2); exit
      ||| QE!spec?tstdt:file[spec=view(spec)
          ,tstdt=edit(oldtd)];exit ) )
 ≫ ( (QE?code:file[code=compile(src1,src2)];
      QE?res:string[res=testcode(code,tstdt)];
      (s_SE(res);exit) )
 ≫ ( ( (r_SE(E);exit) ≫ MakeCode[QE](tstdt) )
      [] ( r_SE(F);exit ) ) )
 ≫ (r_SE(B);exit) )
 [⟩ (r_SE(H);exit)
endproc
```

Protocol entity specification of P1

```
process MakeCode[P1](src1: file):exit :=
 ( ( (r_SE(spec); exit)
 ≫ (s_{P2,QE}(A); exit)
 ≫ (r_{P2,QE}(A); exit)
 ≫ ( (P1!spec?src1[spec=view(spec),
      src1=edit(olds1)]; exit)
     ≫ (s_QE(src1);exit) )
 ≫ ( ( (r_SE(E);exit) ≫ MakeCode[P1](src1) )
      [] ( r_SE(F);exit ) ) )
 ≫ (r_SE(B);exit) )
 [⟩ (r_SE(H);exit)
endproc
```

Protocol entity specification of P2

```
process MakeCode[P2](src2: file):exit :=
 ( ( (r_SE(spec); exit)
 ≫ (s_{P1,QE}(A); exit)
 ≫ (r_{P1,QE}(A); exit)
 ≫ ( (P2!spec?src2[spec=view(spec),
      src2=edit(olds2)]; exit)
     ≫ (s_QE(src2);exit) )
 ≫ ( ( (r_SE(E);exit) ≫ MakeCode[P2](src2) )
      [] ( r_SE(F);exit ) ) )
 ≫ (r_SE(B);exit) )
 [⟩ (r_SE(H);exit)
endproc
```

3.3.2 Derivation system

We have developed the system deriving protocol entity specifications from a service specification automatically. When a service specification in LOTOS, syntax rules of LOTOS (specified in CFG form) and attribute grammars are input to the system, the corresponding protocol entity specifications are output. As described in Section 3.3.1, our derivation algorithm is described in attribute grammars. The system only evaluates these attributes. Since we can modify syntax rules of description language, the class of the language is changeable. Attribute grammars makes us easy to modify the derivation algorithm. Above two facilities are the main characteristics of our system. If the varieties of the operators used in LOTOS are reduced, there are more efficient derivation algorithms which have been proposed(for example, in Ref.[4]). Our system is applicable to those algorithms.

3.4. Evaluation

In the 6th ISPW conference, in order to compare and evaluate the various software process modeling approaches, Kellner et al. have proposed "Software Process Modeling Example Problem" [35] which specifies a process modifying one module of a system. Some solutions about this problem have been reported(see Ref. [11]). We have also tried to describe the service specification of the problem [46]. We have derived the protocol entity specifications from it, and investigated the usefulness of our approach in the following steps.

step1 We have measured the CPU time for deriving the protocol entity specifications from the service specification of ISPW-6 problem with our deriving system and evaluated the result.

step2 We have examined the percentage of the redundant communications in the derived protocol entity specifications.

step3 We have examined the rough ratio between the communications in the protocol entity specifications and the activities in the service specification.

We have used a SUN SPARCstation ELC to measure the derivation time in **step1**.

3.4.1 Evaluation

We have got the result in Table 3.3 and Table 3.4. We have used the simple process description MakeCode in Section 3.2.2 to know the deriving efficiency for a small description.

In **step1**, it has taken 79.7 seconds to derive the protocol entity specifications from the service specification of the ISPW-6 problem whose syntax tree has 3967 nodes(in Table 3.3). This shows that it does not take much time to derive protocol entity specifications with our system for most practical examples.

	Number of nodes of service specification	Derivation time
MakeCode	502	6.2s
ISPW6	3967	79.7s

Table 3.3: Time of deriving protocol entity specifications

	Number of events of service specification	Number of communications in protocol entity specifications
MakeCode	10	47
ISPW6	83	232

Table 3.4: The number of derived communications

In **step2**, we have detected some redundant communication messages in the protocol entity specifications. In the current version of our derivation algorithm, for the service specification such as SE?spec; exit >> P1!spec $\cdots$, the derived protocol entity specifications include two communication messages exchanged between SE and P1. One is a communication for exchanging data *spec* and the other for the synchronization message by the operator >>(in this case the latter communication is needless). The percentage of this kind of redundant communication messages in the protocol entity specifications is less than 10% of all communication messages in the descriptions. Since it seems to be difficult for the process designers to derive the protocol entity specifications from the service specification manually, our derivation algorithm is still useful. Our deriving system which can modify the derivation algorithm easily(in Section 3.3) enables us to improve the algorithm to optimize the communication messages. The details of the optimization are given in Ref. [34].

In **step3**, we have got the result in Table 3.4 where the number of the communication actions in the protocol entity specifications is about 3 to 5 times as many as that of the activities in the service specification. This shows that the communications are troublesome to describe and that they disturb human understanding of the description. So, we are sure that our approach describing no communications in a service specification and deriving its protocol entity specifications automatically is useful to describe and enact software processes.

3.5. Conclusion

In this chapter, we have proposed a LOTOS based technique for deriving protocol entity specifications from a service specification automatically. We have developed the support system which derives the protocol entity specifications and executes each protocol entity specification. We have also applied our approach

to the ISPW-6 problem [35], where we have described its service specification and derived the protocol entity specifications corresponding to the engineers' process descriptions automatically. The derivation time was about 80 seconds with SUN SPARCstation ELC.

Chapter 4

Interactive Execution and Visualization of Protocol Specifications in LOTOS in a Distributed Environment

4.1. Introduction

Generally, in designing phases, the system specifications are frequently modified for debugging and/or improving the systems as the result of the analysis of system behavior. For the efficient behavior analysis, the simulation to execute the specification interactively is useful.

This chapter presents a graphical simulator named PROSPEX(PROtocol SPecification EXecutor) which executes each protocol entity specification and displays its dynamic behavior visually. In order to represent the dynamic behavior of LOTOS specifications with its structural information such as execution dependency between several processes, it is useful to execute the specification and to display the syntax tree of the behavior expression visually. The visual representation of the syntax tree facilitates to observe both the currently executable events and the temporal ordering of events in the future from the current behavior expression. PROSPEX displays such a syntax tree visually, and updates the tree step by step at every event execution. In order to display the dynamic behavior of the specifications effectively, the mechanism for displaying the tree structures appropriately and rapidly. So, a fast layout algorithm provided in Ref. [39] is used to depict such tree structures. In order to facilitate to understand the structure of a large specification, PROSPEX can also enlarge a part of the syntax tree. In this chapter, the design and usefulness of PROSPEX are described.

4.2. Facilities for executing protocol specifications

In this section, we explain the main facilities of our PROSPEX.

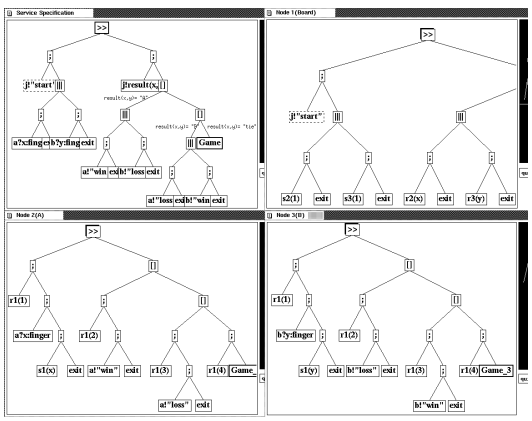
4.2.1 Interactive execution of protocol specifications

In PROSPEX, if a service definition P_S and a protocol specification $<P_1, P_2, \cdots, P_N>$ with N nodes are given, then PROSPEX generates N simulators $S_1, \cdots, S_{N-1}$ and S_N and a monitor S_0. We use each simulator interactively. Each simulator S_k gets the specification P_k of the k-th node as the current behavior expression B_0 when it starts the simulation. Here, the current behavior expression represents the event sequences which the node can execute in the future. The simulator S_k shows which events are executable for the current behavior expression B_i. The user chooses one executable event e from the candidates which the simulator shows. Then, the simulator executes the event e and then computes a new behavior expression B_{i+1} after e is executed. After that, it shows which events are executable for B_{i+1}. The simulation is carried out by repeating these steps.

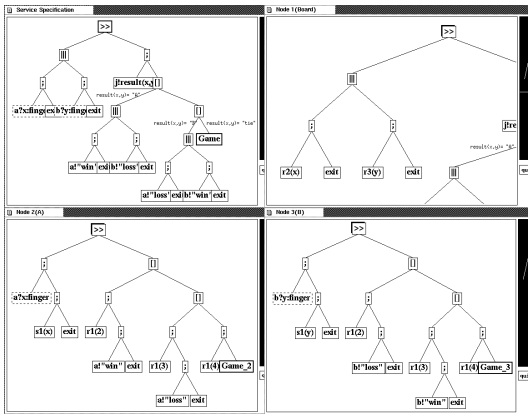
For example, suppose that the service definition in Appendix B and the protocol specification in Appendix C are given. PROSPEX generates three simulators S_1, S_2 and S_3 and one monitor S_0. The monitor draws the syntax tree of the behavior expression of the service definition "Janken". Each simulator S_k draws the syntax tree of the behavior expression of the specification "Janken_k" on a display (see Fig. 4.1(a)).

Each leaf corresponds to either an event, a sending/receiving action or a process name. In Fig. 4.1, the sending/receiving actions "$s_{ij}(m)$" and "$r_{ji}(m)$" at each node "i" are abbreviated as "$s_j(m)$" and "$r_j(m)$", respectively. All executable events are shown by the dotted rectangles. In Fig. 4.1(a), only the event j!"start" at the node 1 is executable. If the user clicks j!"start" at the node 1, then the simulator S_1 executes the event. After j!"start" is executed, a new behavior expression, say B', at the node 1 is obtained. For the new expression B', the sending actions "$s_{12}(1)$" and "$s_{13}(1)$" are executable. The simulator S_1 executes these sending actions automatically without interactions from the user. Then, the receiving actions "$r_{12}(1)$" at the node 2 and "$r_{13}(1)$" at the node 3 become executable. The simulators S_2 and S_3 execute these receiving actions automatically (see Fig. 4.1(b)). In Fig. 4.1(b), the events "a?x:finger" at the node 2 and "b?y:finger" at the node 3 are executable. From Fig. 4.1(b), we can know that both "a?x:finger" and "b?y:finger" are also executable in the service definition. Fig. 4.1(c) denotes that the player A chooses *paper* and inputs it (the input data is assigned on a small window). By executing sending/receiving actions $s_{21}(x)$ and $r_{21}(x)$, the node 1 knows that the player A chooses paper and waits for the data from the node 3 (the player B). If the node 3 sends the choice of player B to the the node 1, then the node 1 calculates the winner and shows it on a small window (see Fig. 4.1(d)). The simulation is continued by repeating the similar interactions.

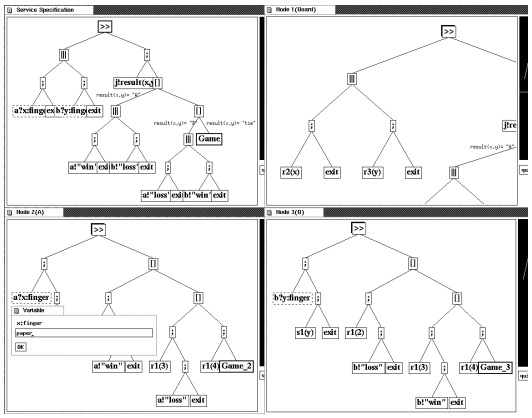
Here, our PROSPEX doesn't draw the labeled transition system(LTS) [31] but the syntax tree of a given behavior expression. By drawing the syntax tree, it is easy to understand the structure of the behavior expression such as the nesting information of parallel and choice operators. We can easily know what kinds of synchronization messages must be received to execute an event. But,



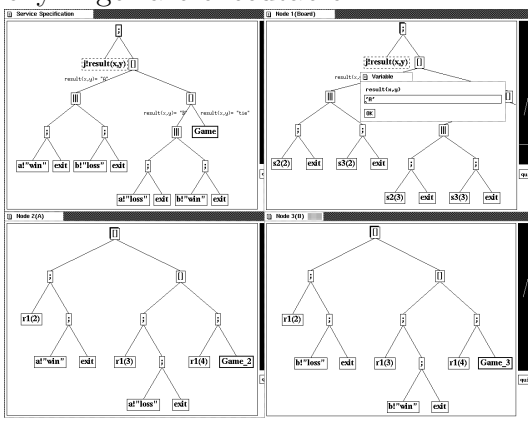
(a) The event j!"start" is executable



(b) The events a?x:finger and b?y:finger are executable



(c) The player A inputs his choice



(d) The Board displays the winner

Figure 4.1: Simulation of protocol specification

if we want to prove the equivalence of two specifications, to draw their LTSs is more suitable. We have developed a test system for LOTOS expressions [38]. In the test system, we have implemented the facility to draw the LTS of a given behavior expression. Now, we are planning to add the facility to PROSPEX.

4.2.2 Observation of correctness of protocol specification for its service specification

PROSPEX has the facility for monitoring a service definition. The monitor S_0 checks a status of the simulation of the protocol specifications against the service definition. If an event on a node is executed and it is not executable for the service definition, PROSPEX indicates to the users that the event cannot be executed. In this case, the protocol specification is not observational equivalent to the service definition. If the events on the service definition are executable, then they are executed automatically according to the information from the corresponding simulator. If the service definition contains nondeterminism, then there may be some problems when an event on a protocol specification is executed. For instance, suppose that the current behavior expression on a service definition is "a;b [] a;c". When the event "a" is executed on a node, the monitor cannot determine to execute which side of event "a" is executed. In such a case, PROSPEX indicates the candidates of the corresponding events, and make the users choose one of them. For each execution step, the users can check whether all executable events in the protocol specification are also executable in the service definition. This facility helps the users to develop correct protocol specifications.

4.3. Design and implementation of the simulator

In this section, we pick up some problems which occurs when we design and implement the support tools such as PROSPEX, and describe how we have solved such problems when we have developed PROSPEX.

4.3.1 Class of LOTOS specifications to be executed

In a LOTOS specification, the temporal ordering of the execution of the events is described as a behavior expression, and the abstract data types used in the specification are described in an algebraic specification language *ACT ONE* [14]. Hereafter, we call the above two parts "the behavior expression part" and "the data type part", respectively. Our PROSPEX supports Full LOTOS functionally although some special operators in Full LOTOS are not supported for simplicity [56].

In the behavior expression part, we can use the following operators: ";", "[]", "||", "|||", "[>", ">>" and "**hide**". But, we do not support the "let", "choice", "par", "accept" and "any" operators. In the data type part, we can use the "type", "library", "sorts", "opns" and "eqns" operators. We do not

support the "formalsorts", "formalopns", "formaleqns", "sortnames" and "opnnames" operators. The axioms in the data type part are treated as a term rewriting system. Therefore , the variables in the right side of each axiom must be appeared in the left side of the axiom. We do not support conditional axioms. In LOTOS, all the abstract data types used in a specification must be described by the users as an algebraic specification even if the abstract data types are primitive. Although some library texts are prepared, some LOTOS simulators cannot use the normal mathematical notations such as decimal arithmetic. This makes LOTOS specifications unreadable. For solving this problem, PROSPEX supports the normal mathematical notations.

If the text "primitives" is described in the library statement, then primitive data types such as integer, character, string, list, array and tuple can be treated as the pre-defined data types. The primitive functions such as if-functions, "+", "−", "∗", "/", "=", ">", "and", "or", "not","car" and "cdr" are also treated as the pre-defined functions. The users do not have to describe the axioms for these primitive functions when they use PROSPEX.

4.3.2 Pre-processes before executing LOTOS specifications

Since, in LOTOS, the function names used in the data type part and their syntax are defined by the users, parsing of LOTOS specifications is not simple. Therefore, some LOTOS simulators do not check the syntax strictly. Here, we give a method to parse LOTOS specifications strictly.

We have defined an algebraic specification language ASL, and developed a support system, ASL system, to design and develop programs in ASL [21, 22]. A specification in ASL can be denoted by a pair $t = (G, AX)$ where G is a context free grammar and AX is a set of axioms. A set of terminal symbols $Term(t)$ generated by G is treated as the terms used in this text. The congruence relation on $Term(t)$ is defined by the axioms AX. ASL system reads a text $t = (G, AX)$ and generates a parser P(G) for G. The parser P(G) examines whether a given term can be generated by G. By using this facility, ASL system examines whether each axiom in AX can be generated by G. If all axioms in AX are generated by G, ASL system generates an interpreter R(AX) which regards each axiom in AX as a rewrite rule, and calculates a normal form of a given term. The normal form corresponds to the value of the given term. Our interpreter R(AX) treats the axioms AX as a term rewriting system. In order to parse a LOTOS specification t_L, PROSPEX divides the LOTOS specification t_L into two parts, the data type part t_D and the behavior expression part t_B. Then, PROSPEX parses each of them.

(1) Parsing Data Type Part

The data type part t_D consists of two sub-parts, the function definition part t_{DF} and axiom part t_{DA}. The function definition part t_{DF} declares the function names used in the axioms and defines their syntax. The axiom part t_{DA} describes some axioms whose functions are all defined in the function definition part. The syntax of the function definition part t_{DF} is defined based on the grammar G_{ACT} of ACT ONE [14, 31]. Therefore, PROSPEX checks the syntax

of the function definition part t_{DF} by using the parser $\mathrm{P}(G_{ACT})$. If the syntax of t_{DF} is correct, then PROSPEX constructs a grammar $G_{t_{DF}}$ for generating the terms consisting of the functions which are defined in t_{DF}. By using $\mathrm{P}(G_{t_{DF}})$, the syntax of the axiom part t_{DA} is checked. If the syntax of both the function definition part t_{DF} and axiom part t_{DA} is correct, then an interpreter $\mathrm{R}(t_{DA})$ is derived.

(2) Parsing Behavior Expression Part

Let G_B be a grammar generating behavior expressions in LOTOS [31]. PROSPEX examines whether the syntax of a given behavior expression is correct by using $\mathrm{P}(G_B)$. The syntax of each guard in the behavior expressions is parsed by $\mathrm{P}(G_{t_{DF}})$.

The parser is developed based on Earley method, it takes $O(n^3)$ times to parse a LOTOS specification whose length is n.

4.3.3 Technique to execute LOTOS Specifications

In this section, we describe the facilities for the execution of LOTOS specifications and their implementation.

(1) Finding Executable Events from Current Behavior Expression

In order to find executable events, PROSPEX searches the syntax tree of the current behavior expression from the root node to the leaf nodes. The way to search a given tree depends upon the operators on the internal nodes. For instance, if the operator ">>" is found while searching a tree, then the next search is continued to its left descendant node. On the other hand, if the operator "[]" is found, the search is continued to the both descendant nodes. Some guard expressions may be described in a behavior expression. While searching tree, if a guard expression is found, PROSPEX calculates the value of the guard by using the interpreter $\mathrm{R}(t_{DA})$ which was explained in Section 4.2. If the value is true, the search is continued. Otherwise, that is, if the value is false, then we do not search the guard expression. Since the value of each guard is stored when the value is calculated, it is not calculated twice.

(2) Execution of Executable Event

The three types of events are supported in PROSPEX: (i) Input events and output events, (ii) Internal events, and (iii) Sending and receiving actions.

The input events and output events are executed by clicking the executable events on the display. When an input event such as "a?x:finger" is clicked, PROSPEX opens a small window. The user can give an input value on the small window (see Fig. 4.1 (c)). When an output event such as "j!result(x,y)" is clicked, PROSPEX calculates the value of "result(x,y)" and display it on the small window (see Fig. 4.1(d)). If a guarded event such as "a?x:int[x>2]" is executed, PROSPEX examines whether the input value satisfies the guard. If the input value does not satisfy the guard, then the execution of the event is canceled. The values of the outputs and guards are calculated by using the interpreter $\mathrm{R}(t_{DA})$.

Internal events are the events which are not seen from the external environments. Usually each internal event is described as "**i**". The "**exit**" event and

the events hidden by the "**hide**" operator are also treated as the internal events. PROSPEX executes these internal events as soon as they become executable.

When a sending (receiving) action is executable, PROSPEX sends (receives) a synchronization message or a data value to (from) the designated node. In order to help the designer to find some redundant synchronization messages, the sending/receiving actions can be also executed in manual. If *manual mode* is selected, then the sending/receiving actions are executed according to the guide from the user. By this facility, the designer can easily recognize how the nodes synchronize each other.

(3) Transformation of Current Behavior Expression

In LOTOS, when an event e is executed for the current behavior expression B, it must be transformed into a new behavior expression B' after e is executed. PROSPEX has the facility for the transformation which was explained in Section 3.

(4) Invocation of A Process

In the syntax tree of a behavior expression which PROSPEX draws, each process is drawn as a node. When some events in the process become executable, the process is invoked. When a process is invoked, it is allowed to replace the gate names and parameter names. Therefore, PROSPEX can replace them. For instance, suppose that "P[a,b](x):= a?y:int;b!x;stop". If P[f,g](2) is invoked, PROSPEX replaces the process P[f,g](2) by the behavior expression "f?y:int;g!2;stop".

4.3.4 Visual display of current behavior expression

Since nondeterminism and parallelism may be described in LOTOS specifications, it is difficult to understand the structure of the current behavior expression if it is displayed as a character string. Therefore, we show the syntax tree of the current behavior expression graphically on a display so that the users can understand its structure at a glance. Since the size of each LOTOS specification may become large, the simulator must be able to handle large trees. Then, we have developed a graph editor **VTM** which can handle large trees with thousands of nodes [39]. VTM makes users easy to observe a whole tree and some specified parts of the tree simultaneously, and makes it easy to input commands for the tree by allowing direct manipulation on the display. VTM is a library program which can be used easily from any application program without the knowledge about X Window System. In order to monitor both the outline of a whole tree and the details of the specified part of the tree simultaneously on a display, VTM prepares two windows, **GlobalView** and **Canvas**(Fig. 4.2).

On GlobalView a whole tree is illustrated where each node is represented by a dot without the label and a rectangle called **AreaMark** are displayed. On Canvas the area specified by AreaMark are zoomed up. Each node on Canvas is displayed by a rectangle in which its label is written. In order to observe several parts simultaneously, it is possible to open more than one canvases. Users can move and resize AreaMark arbitrarily by pushing the center and left mouse

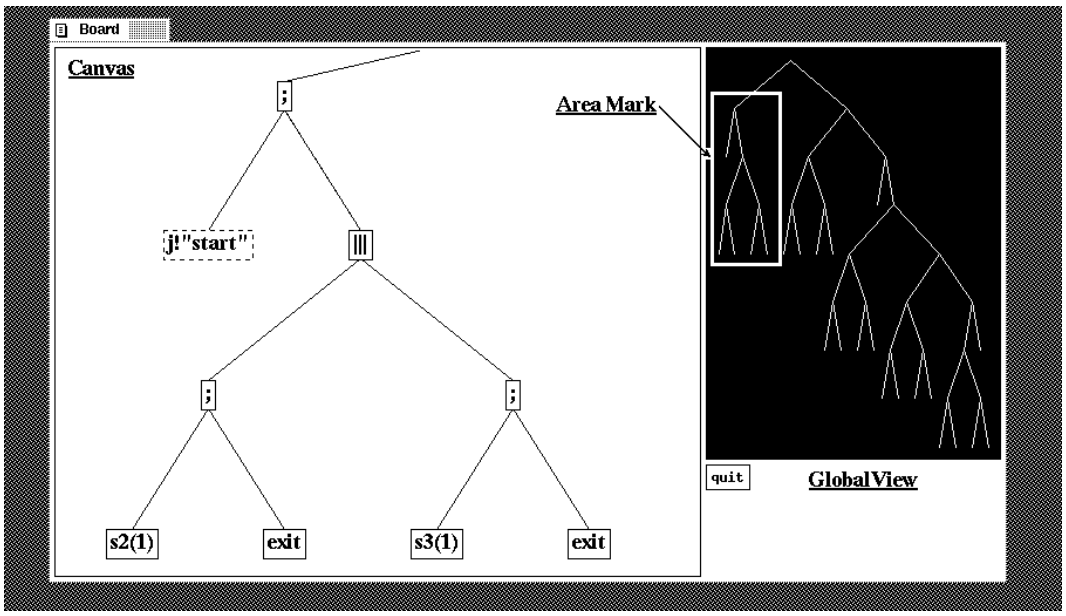


Figure 4.2: Canvas and GlobalView

buttons, respectively, and dragging it to the appropriate directions. It takes less than 0.2 seconds for VTM to calculate the layout and display the whole tree which has 1000 nodes (on a DECstation 3100 with 12MB memory).

4.3.5 Performance of the simulator

In this section, we explain the capacity and the speed of PROSPEX. Generally speaking, the size of the specification PROSPEX can deal with depends on the memory size of the machine where it works. On a DECstation 3100 with 12MB memory, PROSPEX can draw a syntax tree which has thousands of nodes. The number of LOTOS specifications which can be simulated simultaneously depends on the number of file descriptors in UNIX system. About 30 LOTOS specifications can be simulated simultaneously on the DECstation 3100.

We have measured the running time of PROSPEX. In order to execute an input/output event, PROSPEX must (1) check the syntax of the input/output data, (2) calculate a new behavior expression after the event is executed, (3) find executable events for the new behavior expression and (4) draw the syntax tree of the new behavior expression, and so on. For a behavior expression whose syntax tree has about 200 nodes, it takes about 0.5 - 2 seconds to execute the above four tasks. If we must calculate the values of the guard expressions, it takes much time. For Janken Game in Chapter 2, it takes about 6 - 10 seconds for each simulator to execute the LOTOS specification, communicate each other and finish one match.

For the current version of our PROSPEX, there are some limitations. For example, since LOTOSPHERE [16] supports conditional axioms, the inference

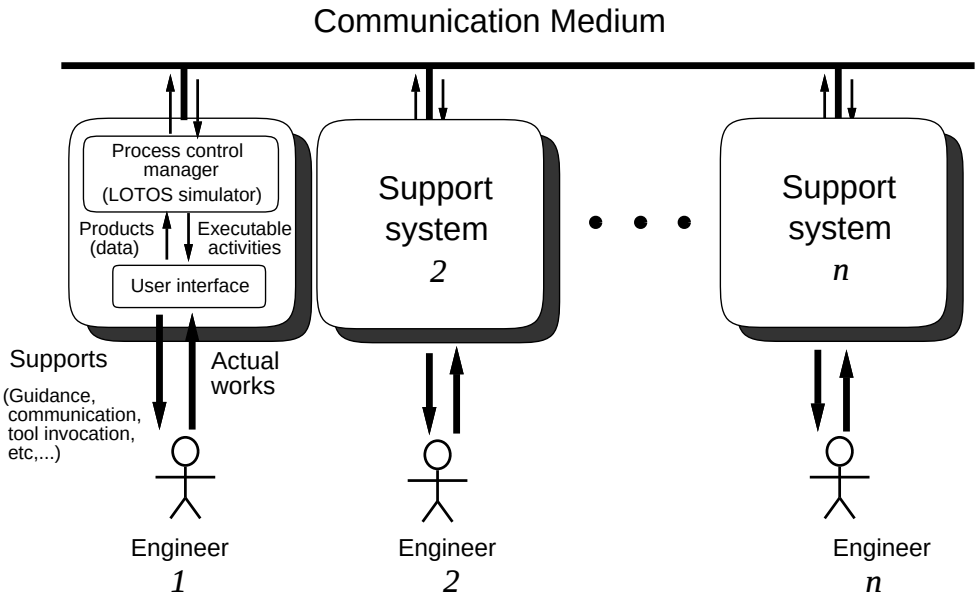


Figure 4.3: Support system

using "narrowing" can be handled. But, our PROSPEX does not support it. Also, our PROSPEX assumes asynchronous communication and cannot handle rendezvous communication among more than two nodes.

4.4. Application

We have developed the support system for enacting the engineers' activities. The composition of the system is shown in Fig.4.3. The description in LOTOS is interpreted and executed with the LOTOS simulator PROSPEX. The process is enacted by executing all individual descriptions in parallel. For each engineer, one system is running with his individual description. All systems cooperate each other to act in order of the whole description, and offer the three main supporting facilities to each engineer. The facilities are (1) Management of the process progress with automatic exchange of communications, (2) Display of the process status, and (3) Guidance of executable activities with automatic activation of tools. We explain them one by one.

• Management of the process progress with automatic exchange of communications

Each support system has the facility for communicating to other systems. Due to this facility, communication massages in the individual descriptions are exchanged among computers automatically and without failures. Therefore, the engineers can focus on their own works.

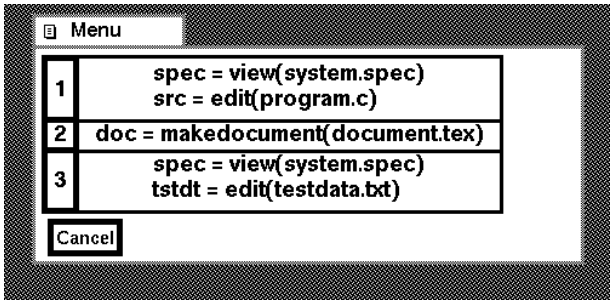


Figure 4.4: Guidance on menu

• Display of the process status

The current expressions of the whole description and individual descriptions in LOTOS make successive transformations. Once an executable activity in the current expression is executed, it is transformed into a new expression in which the executed activity is removed. The syntax trees of the current descriptions which our system displays on a screen (Fig.3.1) enable each engineer to know the current status of the process dynamically. This display of dynamic transformation makes it possible to detect which activities are delayed. The syntax tree of each individual description also informs each engineer in what order the activities are successively getting executable. In general, the size of the syntax tree may become large. For displaying large trees, we have developed a graph editor VTM [39]. VTM makes it easy to observe a whole tree and some specified sub-parts simultaneously. Our system uses VTM for displaying the syntax tree. The users can enlarge and reduce the size of the graph arbitrarily on the display. It takes less than 0.2 seconds for VTM to display a tree which has 1000 nodes (SUN SPARCstation ELC).

• Guidance of executable activities with automatic activation of tools

The support system calculates the executable activities from the current individual description, and shows the contents of them on a menu such as Fig.4.4. When an engineer selects one of them with a mouse, the tools required for the activity are activated. The data required for the activity have been prepared automatically by exchanging messages to the other engineers. Since the failures concerning with data exchange do not exist, the working efficiency of each engineer becomes high.

In general there can be some executable activities at each point of time. These executable activities are hierarchically classified. We call a collection of contents of parallel executable activities a *parallel work set*. We also call a collection composed of some parallel work sets which are selectively executed among them as a *selective work set*. In Fig.4.4, there is a selective work set composed of three parallel work sets 1, 2 and 3. An engineer can select one of them. Suppose he selected 1, then two items $view(system.spec)$ and $edit(program.c)$ in the parallel work set 1 get executable, and the items in other parallel work

sets 2 and 3, get unexecutable. After that, when he selects *view*(*system.spec*) with a mouse, a tool *view* is activated with a parameter *system.spec* and finally the content of *system.spec* is displayed on the display.

Most software development processes include some repetitive processes. In a repetitive process, modification may be applied to the same files again and again. Our support system manages to update files. An engineer can acquire histories of modification about files and information of any version of the files such as modifier, their modification times and so on.

Customization of the system

In our system, the actual programs activated in executing an activity can be changed easily without modifying the formal tool names described in the whole description. Using the *table file*, we can make each formal tool name correspond to one or several actual programs. For example, an activity, "SE?spec:file[spec= edit(...)]", activates a program *emacs* if a tool *edit* is assigned to a program *emacs* in the table file(if the content of the table file is modified to *vi* instead of *emacs*, *vi* will be activated in executing the activity). In order to use a new tool, we only add one line specifying the correspondence between the formal tool name and the actual program to the table file. For flexibility in executing activities, the system also allows us to assign a tool to several programs in the table file. This enables each engineer to use the several tools in executing an activity. For example, if a tool name *makedocument* is assigned to several programs *emacs*, *latex*, *dvi2ps*, *xdvi* and *lpr* in the table file, when an activity "SE?spec:file[spec=makedocument(...)]" is executed, all of these programs are shown on a submenu and each of them can be activated. The engineer can use them repeatedly and in parallel. After his work is finished, he selects "termination" of the activity on the menu. Then a final product of the above process is assigned to a system variable *spec*. The above says the system can be customized as the users want.

4.5. Conclusion

In this chapter, we explained the facilities and implementation of a LOTOS tool PROSPEX which has been developed to observe the execution processes of a tuple of protocol entity specifications and to check whether the the tuple executes the event sequences which satisfy the required service given in the service specification. PROSPEX supports to prove the correctness of a tuple of protocol entity specifications. But, in general, it becomes difficult to prove the correctness if the size of the specification becomes large.

The simulator has been also applied to enact a software process. By executing each protocol entity specification corresponding to engineer's process descriptions at his/her workstation using the simulator, the whole software process has been enacted and the communication between engineers has become automatically exchanged. The simulator was also extended to support each en-

gineer's work by adding the facilities displaying the currently executable events (activities of each engineer) on a menu window, and invoking the required tools for the activity selected on the menu.

Chapter 5

Implementation of LOTOS Specifications using Multi-thread Mechanism and Real-time Visualization of their Execution

5.1. Introduction

At the final stage in developing the distributed systems, the final system specifications must be implemented as efficient object codes for target machines.

This chapter introduces an implementation method for a wide class of LOTOS specifications using a multi-thread mechanism, and a compiler based on the method is provided. The proposed method uses an efficient dynamic message exchange mechanism among processes to implement LOTOS specifications where the number and the combination of synchronizing processes can be only dynamically decided. Abstract data types written in a subset of ACT ONE can be also converted into efficient object codes using the existing compiler for a functional language ASL/F [22, 30].

In general, in order to understand and monitor the dynamic behavior of the real-time systems like communication protocols, it is desirable to display the dynamic behavior of the systems visually using animations. For this purpose, a visualization method for LOTOS specifications is proposed. In the proposed visualization technique, using the multi-rendezvous mechanism of LOTOS, we combine a system specification and its visualization scenario in an extension of LOTOS with synchronization operators concerning with the events to be visualized. A tuple of the original specification and its visualization scenario is converted into the multi-threaded object code by the compiler.

Table 5.1: Target class of behaviour expressions

$B0 =$	$hide\ G\ in\ B1 \mid let\ EQS\ in\ B1 \mid B1$
$B1 =$	$B1 >> B2 \mid B2$
$B2 =$	$B2[> B3 \mid B3$
$B3 =$	$B3\|\|\|B4 \mid B3\|\|B4 \mid B3\|[G]\|B4 \mid B4$
$B4 =$	$B4[]B5 \mid B5$
$B5 =$	$[EXP] -> B6 \mid B6$
$B6 =$	$\alpha; B6 \mid stop \mid exit \mid (B0) \mid P[G](V)$

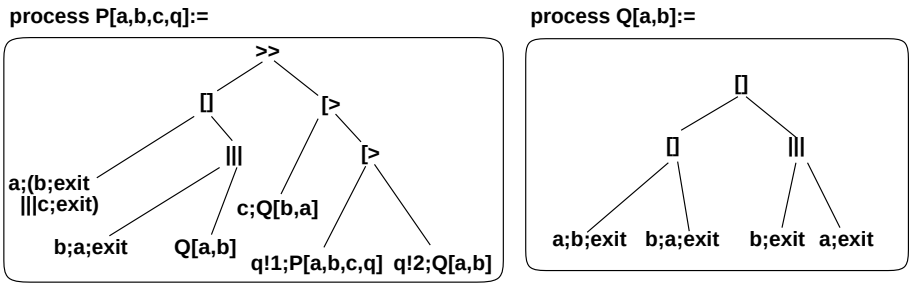


Figure 5.1: Tree representation of a behaviour expression

5.2. Implementation of LOTOS specifications using multi-thread mechanism

5.2.1 Outline of implementation method

In Table 5.1, we show a class of behaviour expressions which we deal with in this paper.

Basic idea to implement LOTOS behaviour expressions

Here, we will explain the basic idea for implementing LOTOS behaviour expressions using our portable thread library PTL [1].

In executing a LOTOS behaviour expression using a multi-thread mechanism, it is desirable to execute each sequentially executable sub-expression of the behaviour expression as a thread independently of other sub-expressions. A behaviour expression can be represented as a tree (Fig. 5.1) where each leaf node is an action-prefixed sequence $\alpha; B$ (here, B is any behaviour expression) or a process instantiation P, and each intermediate node is an operator shown in Table 2.1. In such a tree, when each process instantiation has invoked, its node is replaced by a tree representing the behaviour expression of the process. After process invocations, all leaf nodes will be action-prefixed sequences if there is no infinite invocations specified in the processes. So, we assume each leaf node is an action-prefixed sequence $\alpha; B$ in the below discussion.

In this paper, we implement the behaviour expression by

- mapping each action-prefixed sequence $\alpha; B$ (corresponding to leaf node in Fig. 5.1) to a thread, and
- creating a shared data area whose structure is the same as the part of the syntax tree (corresponding to a sub-tree where each node is an operator in Fig. 5.1).

In order to keep the temporal ordering of events among those action-prefixed sequences, we derive the following object code:

- All currently executable action-prefixed sequences in the behaviour expression are created as threads even if the sequences are alternatively executed.
- Each thread analyzes the shared data area before executing each event and decides whether the event is executable or not. If executable, it executes the event. Otherwise it waits for the event being executable or kills itself (when it need not execute the event).

Let $Code(B)$ be the object code which implements the behaviour expression B. If the behaviour expression B includes enabling operator, say, $B = B_1 >> B_2 >> ... >> B_n$, we derive each object code $Code(B_k)(1 \leq k \leq n)$ in advance, and compose the object code $Code(B)$ so that each $Code(B_k)$ is executed after $Code(B_{k-1})$.

Let B_P be the behaviour expression of process P. If the main process P in the LOTOS specification includes a process instantiation Q and some events in Q are currently executable, we invoke the process Q by connecting the shared data area for B_Q to the corresponding node in the shared data of B_P and executing $Code(B_Q)$.

The structure of the shared data area is dynamically modified since the current behaviour expression changes dynamically when events are executed or the processes are invoked. So, each thread implementing an action-prefixed sequence must be created dynamically and it must analyze the area at every event execution. We call the shared data area used in $Code(B)$ as the *control area* of B, and each currently executable sub-expression as a thread in the current behaviour expression as a *run-time unit*.

Implementation of portable thread library (PTL)

A LOTOS specification includes a number of concurrent processes. In order to derive efficient codes from them, we use a multi-thread library, which can handle concurrent threads (light-weight processes) efficiently within a process. There are several implementations such as LWP within Sun OS [51], the thread library developed in Florida State University [43] and C Threads of Carnegie Mellon University [10] for Mach OS. However each thread library depends on a particular architecture such as Sun OS. For example, since COLOS [13] uses LWP to implement LOTOS specifications, the derived object codes run only on Sun OS.

For deriving efficient and portable codes from LOTOS specifications, we have used Portable Thread Library (PTL) which our research group has proposed in Ref.[1]. The characteristics of PTL are as follows:

(1) All library codes are described only in C language for portability.

(2) Architecture dependent codes are also prepared and used instead of the portable codes if they improve the performance.

(3) Standard application interfaces are provided for general purposes (POSIX 1003.4a [29]).

5.2.2 How to compose object codes for behavior expressions

Here, we will explain how to derive the object code from the behaviour expression B. The derivation consists of three phases: (1) decomposition B into run-time units $\{R_1, ..., R_n\}$, (2) creation of a control area which implements the temporal ordering among run-time units $\{R_1, ..., R_n\}$, and (3) the generation of each object code for R_k.

Decomposition of the behaviour expression into run-time units

Let B be a behaviour expression of a LOTOS specification, and B_1, B_2 be sub-expressions of B, respectively. If B is either $B_1[]B_2$, $B_1|||B_2$, $B_1||B_2$, B_1 $|[g_1, ..., g_n]|$ B_2 or $B_1[> B_2$, we decompose B into B_1 and B_2. The reason why we decompose B even if it is $B_1[]B_2$ is that either B_1 or B_2 may include parallel operators. To the decomposed sub-expressions B_1 and B_2, we similarly decompose them recursively until no more decomposition can apply. Here, if an intermediate sub-expression B' is a process instantiation, we do not decompose it. We define the finally decomposed ones as *run-time units*. Each run-time unit is one of an action-prefixed sequence '$\alpha; B$', a process invocation '$P[g_1, ..., g_n](v_1, ..., v_m)$' and an enabling sequence '$B_1 >> B_2 >> ... >> B_l$'.

For example, in the behaviour expression of process $P[a, b, c, q]$ in Fig. 5.1, the whole behaviour expression is a run-time unit since it is $B_1 >> B_2$ (in Fig. 5.1, let B_1, B_2 be the sub-expressions connected to the left and right side of '>>', respectively). In the sub-expression of B_1, there are three run-time units: (1) a;(b;exit ||| c;exit), (2) b;a;exit and (3) Q[a,b]. The sub-expression B_2 also includes three run-time units: (4) c;Q[b,a], (5) q!1;P[a,b,c,q] (6) q!2;Q[a,b].

Derivation of the control area

Suppose that an alternative execution between two run-time units 'R_1 [] R_2' is given. Then, two threads $Code(R_1)$ and $Code(R_2)$ are created when the object code is executed. What should we do so that either R_1 or R_2 is executed alternatively in the above environment? One of solutions is as follows (see Fig. 5.2).

- The first executed run-time unit (for example, R_1) stores the information that it (R_1) has been already executed to a node corresponding to the operator '[]'.

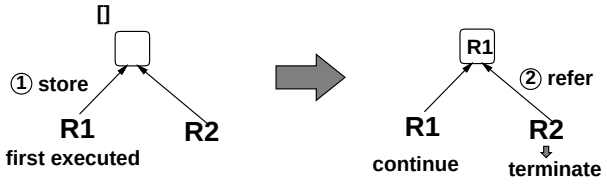


Figure 5.2: alternative execution between two run-time units

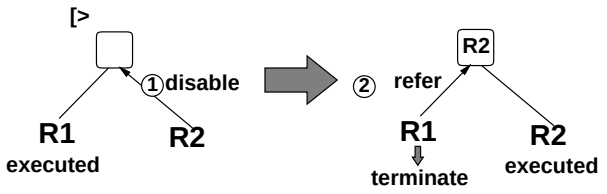


Figure 5.3: interruption by a run-time unit

- When another run-time unit (R_2) is activated, it knows that the above run-time unit (R_1) has been already executed by referring the node and kills itself.

Similarly, $R_1[> R_2$ (where R_2 can disable R_1) can be implemented as follows (see also Fig. 5.3):

- When R_2 is executed, R_2 stores the information about the occurrence of an interruption for R_1.
- R_1 refers the information at every stage of its event execution, and it terminates itself if the occurrence of an interruption is detected. Otherwise it continues its execution.

For the above implementation, we need a node in the control area to keep the following information.

- the information to denote which side of each operator is executed (it denotes which choice is selected for each choice operator ([]) or whether an interruption occurs for each disabling operator ([>) or not)
- the information to denote which side of the operator each run-time unit has connected to.

As mentioned above, the implementation of a behaviour expression composed of two run-time units is quite simple. In general LOTOS behaviour expressions, however, the operators are used hierarchically. For example, the behaviour expression

$$((R_1|||R_2)[](R_3|||R_4))[> R_5$$

means that if R_1 or R_2 (R_3 or R_4) is executed, then R_3 and R_4(R_1 and R_2) must not be executed, and that if R_5 is executed, all run-time units except R_5 must kill themselves (here, we assume R_1, R_2, R_3, R_4 and R_5 are run-time units).

In order to implement general behaviour expressions where the multiple operators are specified hierarchically, (1) the nodes corresponding to the operators should be referred hierarchically from each run-time unit in the control area, and (2) each run-time unit should have the ordered information how it connects to each operator (hereafter, we call the information as a *connection identifier*).

Since the operators ';', '[]', '|||', '|[$g_1, ..., g_n$]|' and '[>' are binary operators, we compose a control area as a binary tree.

In a LOTOS specification S, let B_0 be the behaviour expression of the main process, and B_1, ..., B_h be its sub-processes ($1 \leq h$), respectively (here, the 'process' means a process which is specified semantically as a LOTOS process). Suppose that each B_k includes multiple run-time units (let $R_{k_1}, ..., R_{k_m}$ be such run-time units). For each $B_k (0 \leq k \leq h)$,

(1) parsing B_k to create its binary tree $Tree(B_k)$.

(2) creating the control area $Area(B_k)$ for B_k by removing all $Tree(R_{k_i})(1 \leq i \leq m)$ from $Tree(B_k)$.

(3) for each run-time unit R, if R is an enabling sequence $B_{R1} >> B_{R2} >> ... >> B_{Rn}$, creating each control area $Area(B_{Rk})$.

(4) for each run-time unit R such as 'a;(b;exit ||| c;exit)', if the sub-expression R' of R includes multiple run-time units, creating a control area $Area(R')$ (Fig. 5.4).

By the above steps, all the control areas required to execute the behaviour expression in S can be created statically when we compile each LOTOS specification.

From the behaviour expression in Fig. 5.1, four control areas are generated: $Area(B_{P1}), Area(B_{P2})$ and $Area(b; exit|||c; exit)$ for process P, $Area(B_Q)$ for process Q (Fig. 5.4).

In each node of the control area, we initially put the information about the corresponding operator and the connection identifier, that is, which side the operator is connected to its upper operator.

Generation of the object code for each run-time unit

Each run-time unit R is one of the following three types: (i) a process instantiation ($P[g_1, ..., g_n](v_1, ..., v_m)$), (ii) an enabling sequence ($B_1 >> B_2 >> ... >> B_l$) (where each B_k is any behaviour expression) and (iii) an action-prefixed sequence ($\alpha; B$) (where α, B are any event and any behaviour expression, respectively).

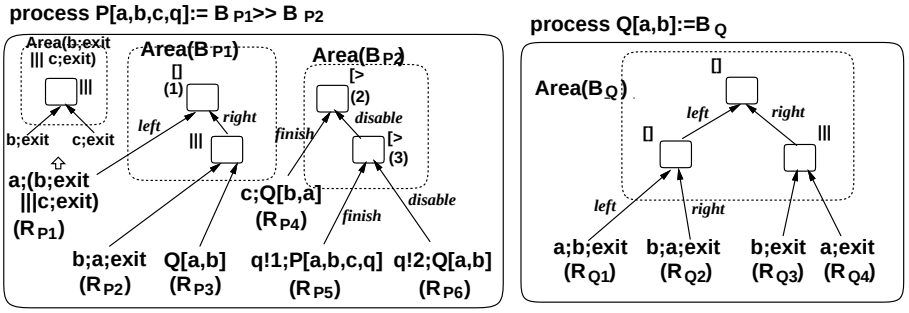


Figure 5.4: Control area

(i) Object code for a process instantiation

When R is a process invocation '$P[g_1, ..., g_n](...)$', all run-time units in the LOTOS process P must be invoked immediately. So $Code(R)$ is as follows:

- connecting the root node of $Area(B_P)$ to the node of the current control area where R has connected.
- invoking $Code(B_P)$.

(ii) Object code for an enabling sequence

If R is $B_1 >> B_2 >> ... >> B_l$, the following object code $Code(R)$ is generated:

(1) connecting $Area(B_1)$ to the current control area in the same way explained above.

(2) invoking $Code(B_1)$

(3) waiting until all run-time units in B_1 finish their execution.

(4) for each $B_2, ..., B_l$, the steps from (1) to (3) are applied.

(iii) Object code for an action-prefixed sequence

If R is an action-prefixed sequence $\alpha; B$, then we generate the following object code $Code(R)$:

(1) $Code(R)$ decides whether it can execute the event α or not by analyzing the current control area.

(2) If $Code(R)$ need not execute the event, it kills itself. Otherwise it waits for the event being executable.

(3) If the event is executable, $Code(R)$ executes the event α after modifying the control area.

(4) If B is also an action-prefixed sequence such as $\alpha'; B'$, then the same operations from (1) to (3) are applied for $\alpha'; B'$ until B is *stop* or *exit*.

(5) If B is not an action-prefixed sequence, $Code(R)$ invokes $Code(B)$ after connecting $Area(B)$ to the corresponding node of the current control area.

The analysis procedure of the current control area in (1) depends on the operators specified in the behaviour expression.

5.2.3 Implementation of LOTOS operators

Here, how all the run-time units are executed by analyzing the current control area so that the temporal ordering of events among them is implemented according to the operators specified in the specification. The analysis procedure of the control area depends on whether synchronization operators are specified or not.

Analysis if choice, parallel and disabling operators are specified

In order to keep which run-time unit has been executed at each choice operator([]), the selected run-time unit stores the marks *left* or *right* to the node corresponding to '[]' in the control area. In order to inform the occurrence of an interruption, the run-time unit which connects to right side of '[>' stores the mark *disable* to the node corresponding to '[>'. The run-time units which connect to the left side of '[>' inform that all the run-time units have finished by storing *finish* to the node of '[>'. We assume that all the nodes in the control area are initially blank.

The run-time unit R can execute its event when one of the following conditions holds at each node from the leaf node (where R connects) to the root node in the control area:

- at each node '[]', the mark *left/right* is stored if R connects to the left/right side of the node (or no mark is stored).
- at each node '[>', the mark *disable* is stored if R connects to the right side of the node (or no mark is stored).

Suppose that the object code for the behaviour expression of the process P in Fig. 5.1 is executed. Since $B_P := B_{P1} >> B_{P2}$, first $Code(B_{P1})$ is invoked and it creates three threads for $R_{P1} = Code(a;\ (b; exit|||c; exit))$, $R_{P2} = Code(b; a; exit)$ and $R_{P3} = Code(Q[a, b])$ with the control area $Area(B_{P1})$ (see Fig. 5.4). Since R_{P3} is a process instantiation $Q[a, b]$, the process is invoked as explained in Sec. 5.2.2 and four threads $R_{Q1} = Code(a; b; exit), R_{Q2} = Code(b; a; exit), R_{Q3} = Code(b; exit)$ and $R_{Q4} = Code(a; exit)$ are created.

Suppose that R_{P1} analyzes the control area first. Since all nodes are blank, it knows it can execute its first event a after it modifies the control area (storing the mark *left* to the node (1) in Fig. 5.4). Then, other run-time units refer the control area and know they cannot execute their events and kill themselves

because another side expression is selected at the node (1). Since R_{P1} executes '$b; exit|||c; exit$' after execution of a, it connects $Area(b; exit|||c; exit)$ to the left side of the node (1) and creates two threads $Code(b; exit)$ and $Code(c; exit)$.

When all threads in $Code(B_{P1})$ have finished, $Code(B_{P2})$ is executed and three threads $R_{P4} = Code(c; Q[b, a])$, $R_{P5} = Code(q!1; P[a, b, c, q])$ and $R_{P6} = Code(q!2; Q[a, b])$ are created to be executed concurrently with the control area $Area(B_{P2})$ as shown in Fig. 5.4. If R_{P4} analyzes the control area first, it executes its event c and invokes the process $Q[b, a]$ since the node (2) in the control area is blank. When an interruption by R_{P5} occurs, the mark *disable* is stored to the node (2). The run-time units $R_{Q1}, ..., R_{Q4}$ invoked by R_{P4} refer the node (2) before they execute their events and know the occurrence of interruption, then kill themselves. Finally, when R_{P6} disables R_{P5}, the mark *disable* is stored to the node (3). R_{P5} detects the occurrence of interruption by referring the node (3) and it kills itself.

The above shows that the original specification in Fig. 5.1 is properly implemented.

Analysis if synchronization operators are specified

LOTOS has highly complicated synchronization mechanism among multiple concurrent run-time units. There are two main reasons for its complexity. The first reason is that the number and/or the combination of the synchronizing run-time units may change dynamically. In Fig. 5.5, for example, R_4 and R_5 may synchronize in executing the event 'b'. However, R_1, R_3 and R_6 may synchronize in executing the event 'a'. The combination cannot be decided statically in advance. The second reason is that I/O parameters must be unified each other. That is, for synchronization, all output values and their sorts of an event must be equal to those of another event, and some proper values must be assigned to all input variables of the synchronizing events. For example, in order that R_2 (its synchronizing event is 'a!4?s:bool') and R_3 (its event is 'a?x:int!true') in Fig. 5.5 synchronize, the proper data '4:int' and 'true:bool' must be assigned to the variables $x : int$ and $s : bool$, respectively.

In general, if a synchronization operator is specified in the behaviour expression $(B_1 \ |[g_1, ..., g_n]| \ B_2)$, the following steps are needed for each run-time unit included in B_1 and B_2:

(1) Before executing each event, each run-time unit examines whether the event must synchronize or not.

(2) If the event must synchronize, the run-time unit keeps the gate name of the event and its output values (and/or input variables) at the operator '|[...]|' and it waits for a synchronization peer. Hereafter, we call the output values and/or input variables as *I/O parameters*.

(3) When another run-time unit executes an event whose gate name is the same as that of the waiting run-time unit, it unifies its I/O parameters and the

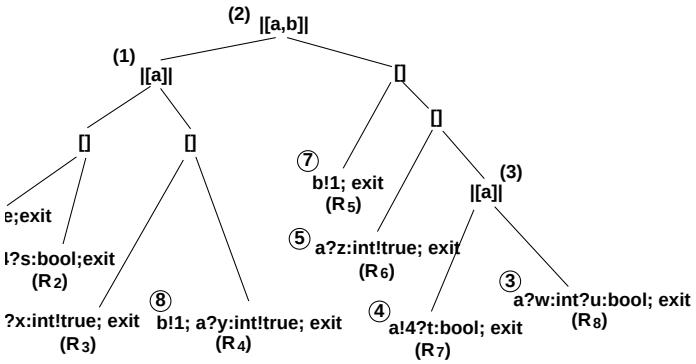


Figure 5.5: A LOTOS specification with synchronization operators

Table 5.2: A synchronization table

gate name	**side**	**state**	**value1**		**value2**		...
a	*right*	–	z	int	true	bool	
b	*right*	*OK*	1	int	–	–	
a	*left*	*NG*	4	int	s	bool	
...							

parameters of the waiting run-time unit by referring the information kept at '|[...]|'.

(4) If it cannot unify its I/O parameters, it also keeps its gate name and I/O parameters at the operator and waits for the peer. When the unification succeeds, the synchronization also succeeds.

(5) If there is no run-time unit which can be the peer, the synchronization fails.

(6) If some synchronization operators are specified hierarchically, the above steps (1)–(5) must be applied to the upper operators in the behaviour expression with the unified I/O parameters.

In order to implement the above, we need a table at a synchronization operator '|[...]|' to keep the following information:

- I/O parameters for the synchronizing run-time units (for unification)
- which sides the run-time units have connected to (for examining whether each waiting run-time unit is a synchronization peer)

For this implementation, we prepare a synchronization table (in Table 5.2) at each node corresponding to '|[...]|' in the control area. There are four items

in the synchronization table : 1. *gate name*, 2. *side*, 3. *state* and 4. *values*. In the item 1, the gate name of each synchronizing event is assigned. In the item 2, the side (*left* or *right*) where the waiting run-time unit has connected is stored. The item 3 denotes the current state of the run-time unit (blank, *READY, OK, NG, RETRY* and *EXEC* are used). In the item 4, I/O parameters and their sorts of the synchronizing event are stored.

We compose the analysis procedure for each run-time unit R (which is $\alpha; B$) of the following three phases: 1. request for synchronization, 2. check of consistency, and 3. propagation of result.

phase 1: request for synchronization

(1) R finds '$|[G]|$' where α's gate name belongs to G by referring the control area from the leaf node. If other alternative run-time units have been executed at an intermediate node '[]' or '[>', R cannot execute α and kills itself.

(2) R finds a peer run-time unit by referring the 'gate name' and 'side' in the synchronization table (it creates a table at '$|[G]|$' if it is not created yet) and tries to unify its I/O parameters and peer's.

(3) If the unification is impossible or no peer is in the table, R adds a line to the table and stores α's gate name and I/O parameters with their sorts to the line. Then R waits until its 'state' turns to *READY* or *RETRY*. If 'state' turns to *RETRY*, R executes the steps from (1) again to find another peer. Otherwise, R works according to the phase 2.

(4) If the unification is possible, R searches the upper nodes in the control area to find the synchronization operator '$|[G']|$' (where α's gate name belongs to G') similarly to (1).

(5) If such a '$|[G']|$' exists, R executes the steps from (2) with the unified I/O parameters.

(6) If such a operator does not exist, a synchronization gets ready and R checks whether all peers are executable according to the phase 2.

phase 2: check of consistency

This phase is to ensure mutual exclusion of several alternative synchronizations.

(7) R stores *READY* to 'state' of the peer line at each '$|[G]|$' and waits a response from each peer. Each peer checks whether other alternative run-time units have been executed or not by referring each node in the control area, then stores *OK* or *NG* to 'state', and waits the result from R.

phase 3: propagation of result

(8) If all peer's responses are *OK*, R informs its peers that the synchronization is successful by storing *EXEC* to 'state' and the unified I/O parameters to 'values' of the peer line at each '|[G]|'. R also stores *RETRY* to other lines in the synchronization table. After all peers receives *EXEC*, then R and its peers get executable.

(9) Otherwise, R stores *RETRY* to 'state' of the peer line at each '|[G]|' and executes the synchronization steps from the phase 1 again.

For example, let us consider how the behaviour expression in Fig. 5.5 is executed in our implementation. Here, we suppose that $R_1, R_3, R_8, R_7, R_6, R_2, R_5$ and R_4 are activated in that order.

First, R_1 creates a synchronization table TBL1 at the node (1) in Fig. 5.5 corresponding to '|[a]|' and stores its I/O parameters $(5 : int, true : bool)$ to TBL1 and waits for the peer. Next, R_3 finds the peer R_1 in TBL1 and tries to unify its I/O parameters $(x : int, true : bool)$ and the I/O parameters of R_1 in TBL1. Since the unification is possible by assigning 5 to the undefined variable x, R_3 refers the upper node (2) '|[a, b]|' with the unified I/O parameters $(5 : int, true : bool)$. Here the synchronization table is not created yet, so R_3 creates the table TBL2 to store the I/O parameters and waits for the peer. Similarly, R_8 creates a synchronization table TBL3 at the node (3) to store its I/O parameters $(w : int, u : bool)$, and waits for the peer. Next, R_7 can unify its I/O parameters and that of the peer R_8 in TBL3 by assigning 4 to w, and refers TBL2 in the upper node (2) with the unified I/O parameters $(4 : int, t : bool)$. Since R_7 cannot unify its I/O parameters $(4 : int, t : bool)$ and that of the peer of R_3 $(5 : int, true : bool)$, R_7 keeps the I/O parameters in TBL2 and waits another peer. When R_6 refers the node (2), it can find the peer R_3 whose I/O parameters $(5 : int, true : bool)$ can be unified to the that of R_6 $(z : int, true : bool)$ by assigning 5 to z. Since there is no upper synchronization operators, R_6 stores *READY* to the corresponding line in TBL2 to check the executability of the peer R_3 (R_3 also stores *READY* to its peer line in TBL1 to check the executability of its peer R_1). Since no other alternative run-time units are executed, R_6 receives *OK* from R_3 (R_3 also receives *OK* from R_1) and informs R_3 that the synchronization is successful by storing *EXEC* to the corresponding line in TBL2 and *RETRY* to the line of other waiting run-time unit R_7 (R_3 also informs R_1 in the same way). Since the nodes corresponding to '[]' are modified before the event $a!5!true$ is executed, other run-time units R_2, R_4 and R_5 detect that other side run-time units are selected by referring the node, and kill themselves.

5.2.4 Implementation of abstract data types

We have developed a functional language ASL/F and its compiler [22, 30]. Using the compiler, we implement the abstract data type (ADT) parts of LOTOS as follows:

(1) converting the ADT part described in ACT ONE [14] into an ASL/F description (it is a syntactical conversion and both languages have similar syntax).

(2) deriving C codes from the ASL/F description.

The derived C codes are compiled and linked to the multi-threaded codes derived from the behaviour expression part explained before.

The functions defined in the ADT parts are used for:

(1) calculating the output values in executing an event (`a!f(x,y)`),

(2) restricting the execution of events by conditions (`a!x[x>0]`),

(3) restricting the execution of behaviour expressions (`[G(x,y)]->B`).

Since these functions are compiled into the functions in C language, we only call each function and get its return value. In (1) and (2), the function is called in executing the event. In (3), $Code(B)$ for the behaviour expression B is invoked if the return value is *true*, otherwise not invoked.

Here, we restrict the class of the ADT parts to a functional program [22, 30]. We need to describe the ADT parts in LOTOS specifications only using the axioms in a functional class.

5.2.5 Optimization

Simplifying control area

In the above explanation, we need $n-1$ nodes in the control area for implementing alternative execution among n run-time units. If we assign different connection identifiers to the alternative run-time units and compose the control area as a general tree, we can reduce the $n-1$ nodes to only a node. We can also remove each node corresponding to '|||' by assigning the same connection identifier to the run-time units executed in parallel. For example, in Fig. 5.6, seven nodes are reduced to a node. Here, we assign the different identifiers to the alternative run-time units $R_1, R_2, R_3(R_4), R_5$ and $R_6(R_7)$ and the same identifier to the run-time units R_3 and R_4 (R_6 and R_7). Our current implementation uses the above technique.

Shortening analysis path in control area and garbage collection

Once a selection or interruption has been executed, some nodes in the current control area may become unnecessary. If we leave such nodes, the analysis path in the control area will be long and it will take much time for each run-time unit to decide whether it can execute each event or not. So we remove such unnecessary nodes from the control area as follows (see Fig. 5.7) :

(1) Once a run-time unit R is selected and the mark (*left/right*) is stored at a node in the control area, the run-time units which connect to the same side of the node skip the node in the next analysis.

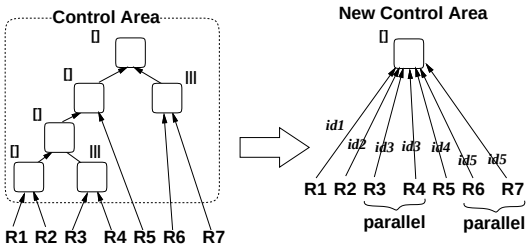


Figure 5.6: Simplification of the control area

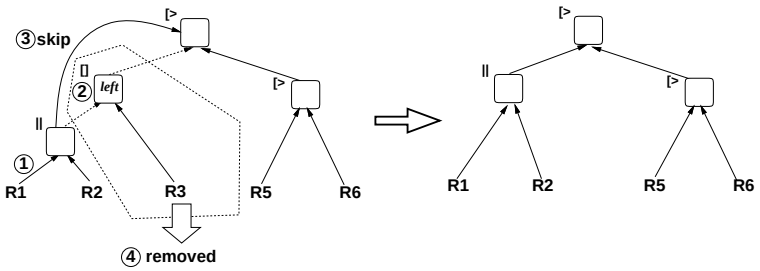


Figure 5.7: Garbage collection of the current control area

(2) Each run-time unit which is not selected removes the unnecessary node before it kills itself.

Local selection among action-prefixed sequences

Alternative execution among multiple action-prefixed sequences can be implemented efficiently within a thread by treating the sequences as a run-time unit and selecting one of alternative events within the unit.

So, we implement such a run-time unit R, say, $a_1; B_1 [] a_2; B_2 [] ... [] a_n; B_n$ as follows (here, a_k and B_k are any event and any behaviour expression, respectively):

(1) $Code(R)$ calculates the set of executable events E from a set of events $\{a_1, ..., a_n\}$ by analyzing the current control area.

(2) If E is empty, it kills itself.

(3) $Code(R)$ selects an event a_i from E and executes a_i after modifying the control area.

(4) If B_i is also alternative execution among action-prefixed sequences such as $a'_1; B'_1 [] ... [] a'_m; B'_m (1 \leq m)$, then the same operations from (1) to (3) are applied for B_i until B_i is *stop* or *exit*.

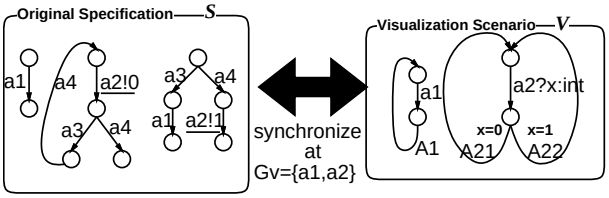


Figure 5.8: Our visualization method

(5) If B_i is not such an expression, then $Code(B_i)$ is invoked with connecting $Area(B_i)$ to the corresponding node in the current control area.

Our current implementation uses the above technique.

5.3. Real-time visualization of dynamic behavior of LOTOS specifications

5.3.1 Visualization method

How to describe scenarios

LOTOS has a multi-rendezvous mechanism[31] which enables multiple concurrent processes to synchronize with respect to the specified gates (events).

For example, two processes P_1 and P_2 can be synchronized with respect to G, a set of gates, describing as follows:

$$P_1 \ |[G]| \ P_2$$

If we specify a gate set {a,b,c} as G, the events executed at those gates are executed simultaneously between P_1 and P_2, and the events not included in G are executed independently. The synchronization mechanism is also called *multi-rendezvous* because the synchronization among more than two processes is possible.

In this paper, we use the multi-rendezvous mechanism to activate the corresponding animation when an event is executed. The basic idea is

- to specify the events and their corresponding animations in a visualization scenario,
- and to execute the original specification and its visualization scenario in parallel using LOTOS multi-rendezvous mechanism.

To visualize the specification S, we describe its visualization scenario V and compose S and V by the synchronization operator as follows (Fig. 5.8):

$$S \ |[G_V]| \ V$$

Here, G_V is a set of gates whose events should be visualized.

Let $S[E]$ be the behavior expression of an original LOTOS specification where E is the set of all gates (events) used in $S[E]$.

If we would like to visualize $S[E]$ with respect to the gate set $G = \{a_1, ..., a_n\} \subseteq E$, we specify the visualization scenario $V[G]$ for $S[E]$ as follows (here, $\prod_{k=1}^{n} V_k$ denotes $V_1|||...|||V_n$):

$$V[G] := \prod_{k=1}^{n} V_k[a_k]$$

$$V_k[a_k] := (a_k; A_k) >> V_k[a_k]$$

Here, A_k is an animation for a_k. If we would like to specify the events in $G' \subseteq G$ so that each event in G' and its corresponding animation can finish before the next event in G' is executed, we modify a part of the visualization scenario as follows (here, $\sum_{k=1}^{n} V_k$ denotes $V_1[]...[]V_n$):

$$V[G] := V'[G'] \;|||\; \prod_{a_k \notin G'} V_k[a_k]$$

$$V'[G'] := \left(\sum_{a_k \in G'} (a_k; A_k) \right) >> V'[G']$$

If we would like to change the scenario $V_1[G]$ to another scenario $V_2[G]$ when an event a_i is executed, we describe it as follows:

$$V[G] := V_1[G - \{a_i\}] \;[> ((a_i; A_i) >> V_2[G])$$

In general, when each event is executed, the data values are input and/or output via the corresponding gate. The data values can be exchanged among the several concurrent processes using the synchronization operator of LOTOS[31]. If we need to change the progress of visualization depending on the data values, we get the values in the visualization scenario using the operator and display the different animation using the values.

Suppose that there is an output event $a!val$ which outputs the value val whose sort is $sort$ via the gate a in the original specification S. If we need to visualize the event $a!val$ depending on the value val, we describe the visualization scenario $V[a]$ for a as follows:

$$V[a] := a?x : sort; \left(\sum_{i=1}^{N} ([cond_i] -> A_i) \right) >> V[a]$$

Here, N is the number of conditions to distinguish the values. A_i is displayed as the animation for a if the condition $cond_i$ holds. "$[cond_i] -> B$" is called a guard expression[31] and it represents that the expression B can be executed only if the condition $cond_i$ holds.

The data from each gate, say a, may have different data types. For example, suppose that the following behavior expression is given.

$$S[a] := a!val_1; ... \;[]\; a!val_2; ...$$

Let us suppose that the types of val_1 and val_2 are "string" and "int", respectively. In order to distinguish the data types and display different animations depending on the data types, for example, we can describe the following visualization scenario.

$$
\begin{array}{l}
V[a] := (\ (a?x : string; A_{11}) \\
\quad [] \quad (a?y : int; (\ ([y < 0] -> A_{21}) \\
\qquad\qquad\qquad [] \ ([0 \leq y \leq 10] -> A_{22}) \\
\qquad\qquad\qquad [] \ ([10 < y] -> A_{23}) \\
\qquad\qquad\qquad) \\
\qquad\quad) \\
\qquad) >> V[a]
\end{array}
$$

Using the mechanism, the data values of each input /output in the specification can be transmitted to the visualization scenario, and different animations can be displayed depending on the values. The above technique can be also used if several values are input/output in an event.

Describing animations

Introduction of animation primitives

In order to describe animations in LOTOS, first we introduce some primitives for animation operations such as registration, indication, movement and elimination of animation objects(*Casts*). Describing each primitive as an event of LOTOS via a special gate for animations, we can compose various animations in combination with those events and LOTOS operators such as parallel, choice and so on.

In this paper, each animation operation is described as follows:

$$AE?id : cast\_t[id = Operation(parameters)]$$
or
$$AE!Operation(parameters)$$

Here, we use AE as a gate for displaying animations on a window called *Stage*. We can also use several Stages simultaneously. If we require n Stages for visualization, we can declare gates $AE_1, \cdots, AE_n$. If we need to distinguish the animation corresponding to each event from others, we can use the gate name associated with the gate in the original specification such as $AE_a, AE_b, \cdots$ (see section of "Structural visualization"). We also use *cast_t* as a sort for an identifier of each Cast, and the identifier for a created Cast is kept in a variable *id*.

For example, we describe an operation which registers a bitmap file "fork.xbm" as a Cast used in an animation as the following event.

$$\text{AE?fkid : cast\_t[fkid = CreateCast("fork.xbm")]}$$

We show the part of animation primitives in Table 5.3.
Using the operators in Table 2.1, we can describe various animations such that the multiple Casts are moving in parallel.

Table 5.3: Animation primitives

Primitives	**Contents**
CreateCast	registering a bitmap as a Cast
CreateString	registering a string as a Cast
CopyCast	creating a copy of a Cast
MoveCast	moving a Cast to the specified location in the specified time
DestroyCast	destroying a Cast
ChangeAttribute	modifying the attribute of a Cast
...	...

Structural visualization

In LOTOS, it is recommended that the specifications are described in the resource oriented and/or constraint oriented styles[55]. Most of existing LOTOS specifications are hierarchically described where the processes for specifying the behavior of resources and the processes describing the restrictions among them are executed in parallel. In visualizing such specifications, we would like to see a part of behavior such as actions to the external environment as well as the whole behavior.

In our visualization method, the animations for the events hidden by **hide** operator of LOTOS are not displayed on the display.

For example, for the original specification S[a,b] and its visualization scenario V[a,b], the visualized specification VS[a,b] is described as follows:

$$\mathrm{VS[a,b]} := \mathrm{S[a,b]} \ \ |[\mathrm{a,b}]| \ \ \mathrm{V[a,b]}$$

If we want to see only the animations for event "a", we add **hide** operator as follows:

$$\textbf{hide}\ \mathrm{b}\ \textbf{in}\ \mathrm{VS[a,b]}$$

In order to hide the animation events depending on the events, we use several animation gates such as AE_a and AE_b for describing the animations for events "a" and "b". By describing "**hide** b **in** ...", the event AE_b is also hidden. The details are described in Ref. [58].

5.3.2 Execution of visualized specifications

We use the LOTOS compiler [59] explained in Sec. 5.2 to convert the visualized specifications into the executable object codes. Here, we have extended the LOTOS compiler to treat the animations. We have used our Interactive Animation Server[50] to display the animations in the generated object codes. Since most of the animation primitives defined in Table 5.3 are equivalent to

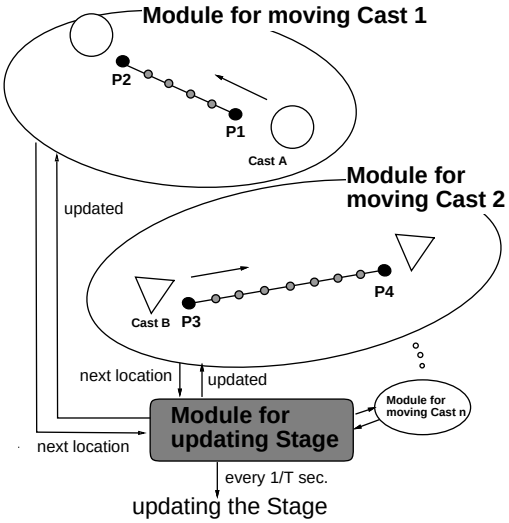


Figure 5.9: Mechanism for real-time animations

the instructions on Animation Server, the LOTOS compiler only replaces the animation primitives to the subroutine calls in C language.

Mechanism for real-time animation

Our Animation Server[50] enables an easy operation for each animation object(*Cast*) such as registration, indication, modification of attributes (*i.e.* location, color, priority, and so on) and elimination of it. To indicate animations successively according to a scenario using Animation Server, it is needed to repeat the following process:

- informing the instructions for changing Cast attributes to Animation Server in advance.
- updating the animation window (*Stage*) so that all modifications for Cast attributes are reflected.

For easy construction of the visualization scenarios, we allow the primitive operation which enables each Cast to move to the destination in the specified time. Since LOTOS allows parallel execution of multiple processes, the mechanism that the multiple Casts can move in parallel is needed. For this purpose, the Stage for Casts should be updated at a fixed time interval.

We have composed the mechanism of two types of modules: a module for updating Stage and a module for moving each Cast. For simplicity, we fix the time interval for updating the Stage to a certain constant such as 30 frames/sec (here, we call each image displayed on the Stage at every time interval as *frame*).

When the multiple modules for moving Casts are executed in parallel, they and a module for updating Stage are synchronizing at a fixed time interval as the following steps (Fig. 5.9).

[Behavior of a module for moving each Cast]

1. calculating the location at the next frame from the interval for updating, the distance to the destination and the time for whole movement and calling the instruction of Animation Server for modifying the location of Cast.

2. waiting until the Stage is updated and repeating the above process until reaching the destination.

[behavior of a module for updating Stage]

1. calling the instruction for updating the Stage of Animation Server at every fixed time interval.

Since a module for updating Stage and all modules for moving Casts are mapped to threads in the generated code, so they run fast. Using the mechanism, the dynamic behavior of the multiple concurrent processes can be visualized.

5.3.3 Example of visualization

In order to examine that our visualization method is useful to understand the dynamic behavior of concurrent systems, we have tried to visualize a LOTOS specification. We have selected Dijkstra's dining philosophers as an example of a concurrent system.

Dijkstra's dining philosophers

"Dijkstra's dining philosophers" is a typical model for discussing problems arising in concurrent systems. In the model, five philosophers (processes) are acting concurrently, either thinking or eating. Five philosophers are sitting around a table, and one fork (resource) is placed between any two neighbor philosophers. Each philosopher must take two forks in his both hands to eat.

In order to enable all philosophers to cooperate to proceed without deadlocks, a control for accessing the shared forks fairly is required.

Specification of philosophers

In order to represent the actions of each philosopher, we introduce the following events in LOTOS:

lfk!take!h_to_r	philosopher takes his left fork 'lfk' ('lfk' is moved from the home location to the right side)
lfk!release!r_to_h	philosopher releases his left fork 'lfk' ('lfk' is moved from the right side to the home location)
rfk!take!h_to_l	philosopher takes his right fork 'rfk' ('rfk' is moved from the home location to the left side)
rfk!release!l_to_h	philosopher releases his right fork 'rfk' ('rfk' is moved from the left side to the home location)
ph!eat	philosopher 'ph' eats
ph!think	philosopher 'ph' thinks

Note that each action concerning with a fork 'lfk/rfk' has enough information for both a philosopher 'ph' and the fork 'lfk/rfk'.

Using the above actions, we have described a behavior of each philosopher as the following process in LOTOS:

```
process Philosopher[ph,lfk,rfk]: noexit:=
( ph!think; exit
[]
  lfk!take!h_to_r;(
        rfk!take!h_to_l; ph!eat;
          rfk!release!l_to_h; lfk!release!r_to_h; exit
     [] lfk!release!r_to_h; exit )
) >> Philosopher[ph, lfk, rfk]
endproc
```

In the specification, each philosopher starts thinking or trying to take his left fork. If he can take it, he tries to take his right fork. If he can take it, then he starts eating. Otherwise he releases his left fork to avoid deadlocks.

We describe the specification of each fork process similarly as follows:

```
process Fork[fk] : noexit :=
(  fk!take!h_to_l; fk!release!l_to_h; exit
[] fk!take!h_to_r; fk!release!r_to_h; exit
) >> Fork[fk]
endproc
```

The above specification represents that after the fork 'fk' is moved from the home location to the left/right side, it must be moved from the same side to the home location.

According to the above discussion, the specification for all philosophers is described by combining five philosopher processes and five fork processes using the parallel and synchronization operators of LOTOS. We use the gates ph1, $\cdots$, ph5 and fk1, $\cdots$, fk5 for distinguishing five philosophers and five forks, respectively (here, fk1 is shared between ph5 and ph1, fk2 between ph1 and ph2, $\cdots$).

```
process Philosophers[ph1,...,ph5, fk1,...,fk5]
   : noexit :=
  (Philosopher[ph1,fk1,fk2] ||| ...
||| Philosopher[ph5,fk5,fk1])
|[fk1,...,fk5]|
  (Fork[fk1] ||| ... ||| Fork[fk5])
endproc
```

Visualization scenario

In visualizing specification "Philosophers", we keep displaying the animation objects for five forks and five philosophers throughout, and activate the corresponding animation when a particular event in "Philosophers" is executed.

When an event "a philosopher taking a fork" is executed in "Philosophers", we would like to display the following animation:

- moving the fork object from its home location to the philosopher (moving for the opposite direction in "releasing")

Similarly, we would like to display the following animation when an event "a philosopher thinking (or eating)" is executed.

- changing the shape of the philosopher object to the corresponding one (either "thinking object" or "eating object") for a certain time.

In order to describe the visualization scenario, first we define the animation objects (Casts). Here, we define the Casts for five forks and five philosophers. For example, a Cast for "fork1" is defined using a primitive in Table 5.3 as follows:

$$\text{AE?fkid1} : \text{cast\_t}[\text{fkid1} = \text{CreateCast}(\text{"fork1.xbm"})]$$

(here, "fork1.xbm" is a name of a bitmap file of "fork1")

In the next step, we describe the animations in LOTOS.

We describe one process which animates each moving fork. Let *home* be the home location of each fork. Let *left* be the destination of the moving fork when the left side philosopher takes it (let *right* be the destination when the right side philosopher does). In order to select the destination of the fork, we describe the visualization scenario for forks so that it can know which side philosopher takes/releases the fork. The following is an example of the visualization scenario for forks. Here, we specify that each fork animation takes t seconds.

```
process MvFork[fk,AE](home,left,right:pos_t,fkid:cast_t)
 : noexit :=
(
   fk!take?dir:direction_t;
   ( [dir=h_to_l]->AE!MoveCast(fkid,home,left,t);exit
  [] [dir=h_to_r]->AE!MoveCast(fkid,home,right,t);exit
   )
[] fk!release?dir:direction_t;
```

```
   ([dir=l_to_h]->AE!MoveCast(fkid,left,home,t);exit
[]  [dir=r_to_h]->AE!MoveCast(fkid,right,home,t);exit
   )
) >> MvFork[fk,AE](home,left,right,fkid)
endproc
```

Other actions of philosophers "eating" and "thinking" are visualized similarly by describing their visualization scenario MvPhilo[ph,AE](...).

The whole visualization scenario is described as follows:

```
process VS[fk1,...,fk5,ph1,...,ph5,AE]:noexit :=
( Definitions of all Casts ) >>
(   MvFork[fk1,AE](HOME1,LEFT1,RIGHT1,fkid1)
||| ...
||| MvFork[fk5,AE](HOME5,LEFT5,RIGHT5,fkid5)
||| MvPhilo[ph1,AE](...)
||| ...
||| MvPhilo[ph5,AE](...)
)
endproc
```

(here, the constants $HOME_n$, $LEFT_n$ and $RIGHT_n$ represent the home location and the left/right destinations of the fork n, respectively)

The original specification "Philosophers" can be visualized by combining it and its visualization scenario "VS" with the synchronization operator as follows:

$$\text{Philosophers} \ |[\text{fk1}, ..., \text{fk5}, \text{ph1}, ..., \text{ph5}]| \ \text{VS}$$

We have executed the visualized specification of philosophers in our system explained in Sec. 5.3.2. The animations are displayed on the graphic window (Fig. 5.10). Fig. 5.10(a) shows the initial state. Fig. 5.10(b) shows the situation that the left-down side philosopher is now taking his right fork after he took his left fork. And the top and right-down side philosophers are now thinking. The right-up side philosopher has just taken his left fork. In Fig. 5.10(c), the left-down side and right-up side philosophers are now eating after taking two forks. The left-up side philosopher has taken his left fork and is trying to take his right fork. Other philosophers are thinking. In Fig. 5.10(d), the top side philosopher is eating after he took two forks which had been taken by both side philosophers. And left-down side philosopher is now releasing his right fork. Other three philosophers are thinking.

Like the above, we can easily understand the dynamic behavior of concurrent processes from the animations.

5.3.4 Related work

There are several researches for visualizing protocol specifications for facilitating understanding and designing them and for a stepwise refinement[3, 53]. In formal specification language Estelle, each process is described as a finite state machine. In Ref. [3], a visualization technique of Estelle specifications and its

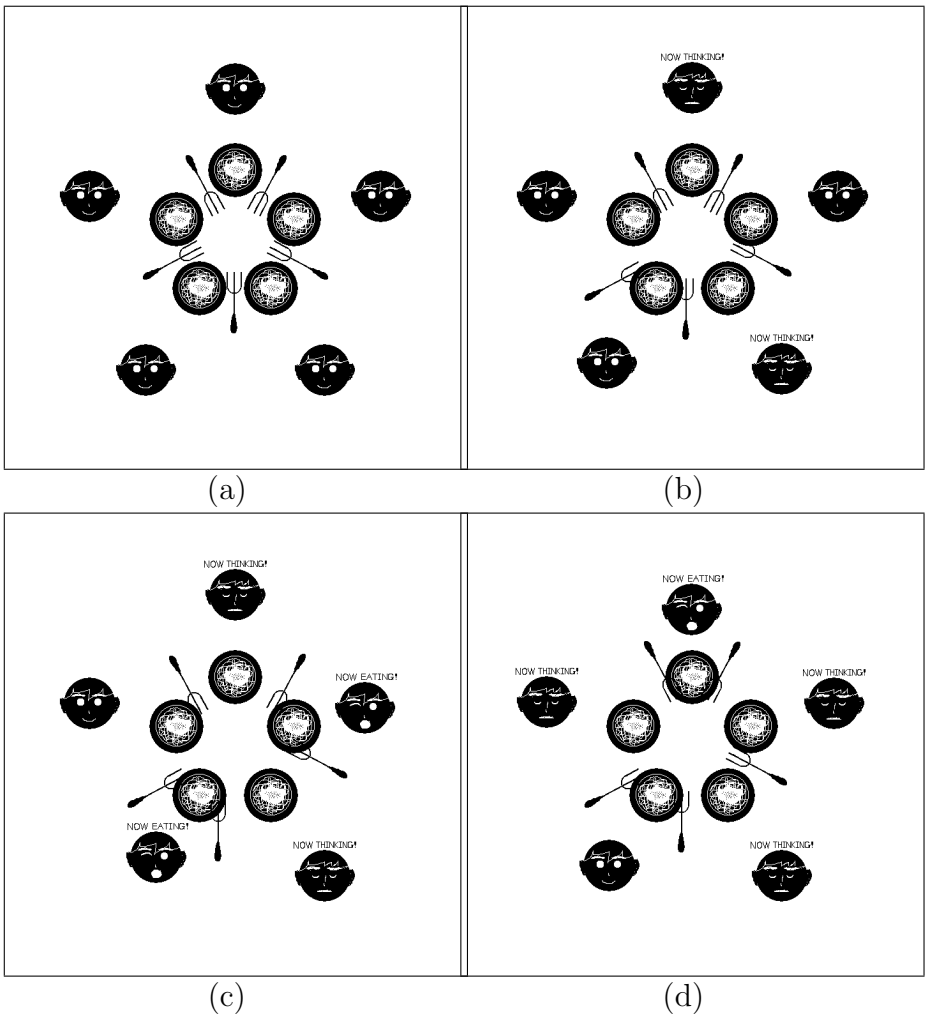


Figure 5.10: Visualized Dijkstra's philosophers

Table 5.4: The number of executed events per second in parallel execution

Number of run-time units	Ours	COLOS	TOPO
100	1724/sec	700/sec	57/sec
200	1214/sec	421/sec	26/sec
300	928/sec	276/sec	17/sec
400	712/sec	227/sec	12/sec
500	565/sec	183/sec	9/sec

system called GROPE are proposed. GROPE reads a given Estelle specification and displays its state machines to the graphic window, where the state transitions are dynamically visualized. In GROPE, each process as a black box is also displayed as a rectangle. Communications between two processes are implemented so that the animation objects representing messages are moving along the line drawn between two processes (rectangles). In GROPE, since the specification is executed virtually by an interpreter, the visualization makes an interactive progress. It is useful for understanding of the behavior of protocols, but the visualized specifications may run much slower than the programs derived from the specifications using compilers. SOLVE [53] is proposed as a visual language based on LOTOS instead of G-LOTOS [32] which is a graphical representation of LOTOS. SOLVE aims at facilitating the design and description of LOTOS specifications using visual and easy operations like animations. Using SOLVE, the designers can describe the system specifications only using interactive operations, if they do not know LOTOS. The specifications described in SOLVE can be converted into LOTOS specifications. However, it also uses the simulator to execute the visualized specification, so the real-time visualization of concurrent systems may be impossible.

5.4. Evaluation

5.4.1 Evaluation of implementation method

Experimental results

Here, we will evaluate our implementation method. Although some LOTOS compilers have been developed for several years, most of them do not use multi-thread mechanisms to implement LOTOS specifications. As long as we know, only COLOS [13] uses a multi-thread library to implement LOTOS specifications efficiently. COLOS has been developed within the ESPRIT project LOTOSPHERE and is based on the algorithm introduced in Ref. [13]. We have compared our compiler with other compilers, COLOS [13] and TOPO [40, 41].

In order to examine how efficiently multiple concurrent processes are executed, we have measured the number of events executed in one second when we execute the object codes derived LOTOS specifications which include several hundreds of concurrent run-time units. In the specifications, all run-time units

Table 5.5: The number of executed events per second in alternative execution

Number of Alternatives	Ours	COLOS	TOPO
5	1900/sec	510/sec	600/sec
10	940/sec	520/sec	360/sec
15	730/sec	510/sec	240/sec
20	570/sec	516/sec	170/sec

are connected by the parallel operator '|||', and each run-time unit is an action-prefixed sequence composed of ten events. We show our experimental results in Table 5.4. Here, we have used Sun SPARCstation IPX with 24MB memory.

Table 5.4 shows that our compiler is more efficient than COLOS and TOPO in concurrent execution. Since TOPO do not use a multi-thread library, the object codes by TOPO run much slower than others with respect to parallel execution. In our compiler, more than a thousand of concurrent run-time units can be derived and executed fast if we assign 8KB memory to each thread stack (on Sun SPARCstation IPX with 24MB memory).

In order to examine the efficiency in alternative execution, we have measured the number of events which are executed in a second for the following process P:

$$P := (a_1; exit[]a_2; exit[]...[]a_n; exit) >> P$$

In measurement, we have changed the number of alternatives n from 5 to 20 for examining the selection cost for the case that many alternatives are specified. We show the result in Table 5.5. In the experiment, the object codes derived from our compiler and TOPO have selected each alternative in each loop. Otherwise, the code from COLOS has always selected the first alternative.

In LOTOS, it is recommended that the specifications are described in resource oriented and/or constraint oriented styles [55], and a lot of LOTOS specifications have been described in those styles. In order to examine how efficiently the specifications in those styles are executed, we have also measured the number of events in one second when we execute the specifications with several constraints. For the behaviour expression B which includes 100 run-time units connected by '|||', we have used the specifications $B||B$ (one constraint), $B||B||B$ (two constraints), $B||B||B||B$ (three constraints) and $B||B||B||B||B$ (four constraints) for measurement. We show the experimental results in Table 5.6.

According to Table 5.6, the codes generated from COLOS run faster than others with respect to the synchronization. However, COLOS restricts the class of LOTOS behaviour expressions. For such a restricted class, more efficient implementations for synchronization may be considered on our implementation.

Table 5.6: The number of executed events per second in synchronous execution

Behaviour expression	**Ours**	**COLOS**	**TOPO**
B	1724/sec	700/sec	57/sec
$B\|\|B$	244/sec	509/sec	12/sec
$B\|\|B\|\|B$	195/sec	365/sec	0.9/sec
$B\|\|B\|\|B\|\|B$	175/sec	273/sec	—
$B\|\|B\|\|B\|\|B\|\|B$	155/sec	224/sec	—

Related work

Several LOTOS compilers have been proposed and they are classified into four approaches. The first approach is to derive a finite state machine from a given LOTOS specification by reducing parallelism in the specification [17]. However, this approach focuses only the resource oriented specifications, so we cannot deal with the general LOTOS specifications. The second approach is the technique to use the languages which can handle parallel processing such as PARLOG [18]. In this approach, each parallel process can be easily mapped to a concurrent unit in those languages. However, the efficient implementation of synchronization and interruption among multiple concurrent processes is difficult due to the difference between LOTOS and those languages. Third approach uses the technique for mapping each concurrent process to a UNIX process [8]. Although it implements the concurrency, the overhead in the context switching and communication among processes causes low performance in the derived codes when there are many concurrent processes in the specification. The last approach uses the mechanisms to handle concurrent processes efficiently [13, 40, 41, 47]. In Ref. [40, 41], a LOTOS specification is transformed into an abstract model which is independent of machine architectures. However, in its current implementation, co-routine calls of concurrent processes are virtually executed. So, the generated codes are not very fast. In Ref. [47], a process scheduler written in an assembly language is used for executing concurrent processes. A LOTOS compiler COLOS [13] uses SUN's light-weight process mechanism(LWP) [51] for this purpose. However, those mechanisms depend on machine architectures and the generated codes may not be executed on various machines and/or operating systems.

Most of existing LOTOS compilers except TOPO [40, 41] cannot generate the object codes automatically from the abstract data type (ADT) parts of LOTOS specifications. Although COLOS has a framework handling the ADT externally, the contents of the ADT functions must be described in C language by the designers. Our LOTOS compiler can generate the object codes from the ADT parts automatically by using the compiler for our functional language ASL/F [22, 30].

Since existing LOTOS compilers [13, 47] use hardware dependent mechanisms for generating fast object codes, derived object codes may not be portable.

Table 5.7: Amount of description in C and LOTOS

Lang.	Orig. Spec.	Scenario	Total
C	—	—	288 (steps)
LOTOS	23 (steps)	37 (steps)	60 (steps)

Using our implementation method proposed in this paper, we can derive fast and portable object code from LOTOS specifications. According to the above, our LOTOS compiler can be used more widely to develop actual systems and protocols.

5.4.2 Evaluation of visualization method

In order to examine (1) the facility for describing visualization scenarios and (2) overhead costs for visualization in our method, we have carried out some examinations.

For evaluating (1), we have also described a program implementing the same visualization of Dijkstra's philosophers (explained in previous section) in C language and compared with the description in LOTOS. The program in C uses the primitives in Table 5.3 and the multi-thread library as well as the code which the LOTOS compiler generates.

The comparison about the amount of description in C and LOTOS is shown in Table 5.7. Here the number in the LOTOS specification represents the sum of events and process invocations. We also used 25 LOTOS operators to specify alternative, parallel and synchronization execution.

The reason why the program in C is larger than one in LOTOS is because the concurrent execution of multiple processes must be sequentially described in C even if we use the multi-thread libraries. In describing the program in C, it was difficult to compose the animation part independently of the control part since C has no synchronization mechanism as a standard. In LOTOS, we can describe the visualization scenario easily and independently of the control part.

For examining (2), we have measured the number of executed events per second in both cases of executing the original specification and the visualized specification in Sec. 5.3.3 (here, we do not take the time for animations into account). The number of executed events per second in the visualized specification was about 80% of the number in the original specification. The percentage varies depending on the number of events to be visualized. The reason why the execution is slow down in the visualized specification is that the processes for the animation are added and all events must synchronize between the visualization scenario and the original specification. In the generated codes from the LOTOS specifications, more than 150 events are executed at every second.

5.5. Conclusion

In this chapter, we have described an implementation method of LOTOS specifications using our portable multi-thread mechanism. Our compiler based on the method can treat all of the basic operators in LOTOS such as choice ([]), parallel (|||), synchronization (|| and $|[g_1, ..., g_n]|$), enabling (>>) and disabling ([>). It can also treat the ADT parts described as functional programs in LOTOS specifications. Since the derived object codes are portable, they can be executed on many architectures and/or OSs.

According to the results in Sec. 5.4, we believe that our compiler can be used for the developments of many practical distributed systems and communication protocols more widely.

The current version of our compiler does not support (1) `par`, `choice` and `accept` statements, nor (2) the parameterized `exit` operator. Now we have been extending our LOTOS compiler so that they can be used. Our current implementation cannot generate faster object codes for the restricted class of the LOTOS behaviour expressions than the compilers which can generate efficient object codes for such a restricted class. We would like to improve our compiler to generate optimized codes from the specifications written in such a class. The developments of practical communication protocols using the compiler is one of our future work.

We have also proposed a method for visualizing LOTOS specifications using the multi-rendezvous mechanism.

Main characteristics of our visualization method are that (1) the dynamic visualization depending on the contents in the given specification becomes possible since the visualization scenario is also described in LOTOS, that (2) the visualization scenario can be described without modifying the original specification, and that (3) the real-time visualization of concurrent systems becomes possible. We could visualize the LOTOS specification of Dijkstra's philosophers based on our method with a lower cost. From the experimental results, we can execute the visualized specification as fast as the original specification.

In our visualization method, only the gate names and the values of each executed event in the original specification are used in the visualization scenario. Suppose the several parallel processes can execute a same event whose gate name and values are the same. Under the situation, to display a different animation depending on a process which executed the event, we need to modify the original specification so that the events which belong to those processes can be distinguished. In our visualization method, each animation synchronizes the event in the original specification only at the start point. If we need to synchronize the end point of the event, we modify the original specification.

In our current system, we must prepare each animation description in LOTOS for visualization. To compose the visualization scenario more easily, we are now trying to design and implement an interactive tool for making animation behaviors on a graphical window by mouse operation.

Chapter 6

Conclusion

In this thesis, as a method to design and develop reliable distributed systems efficiently, we have studied the following three research topics on the execution and visualization of the system specifications in a wider class of LOTOS.

(1) For a given service specification with data parameters and all basic operators in LOTOS, and an assignment of each gate to the node, a technique to derive correct protocol entity specifications has been proposed. A derivation system based on the algorithm has been also developed.

(2) A graphical LOTOS simulator which can execute a tuple of protocol entity specifications and display the dynamic behavior visually, has been provided.

(3) An implementation method for a wide class of LOTOS specifications using a multi-thread mechanism and a compiler based on the method has been proposed. A real-time visualization method for LOTOS specifications using the compiler has been also proposed.

It was shown that the derivation technique can be applied to a practical service specification with data parameters and the operators such as synchronization and interruption using the ISPW6 problem as an example. It was also shown that the proposed LOTOS simulator can be used to analyze the behavior of distributed systems. The experimental results have shown that the proposed compiler can generate efficient object codes from a wider class of LOTOS specifications where the tuple of synchronizing processes is dynamically decided, and that the compiler can also generate object codes which can display the dynamic behavior of parallel processes in LOTOS specifications in real time.

As explained in 3.3, the proposed derivation method assumes the asynchronous message exchanges to assure the temporal ordering of events and data distribution among the distributed nodes. However, LOTOS has multi-rendezvous (synchronization) operators which enable multiple concurrent processes to synchronize in executing events and to exchange data values. If we can use the operators for the communications among the nodes, we can describe

each protocol entity specification more simply and clearly. Although the proposed derivation algorithm can be easily modified to use such multi-rendezvous operators, it is still difficult to implement the multi-rendezvous mechanism efficiently on a distributed environment with asynchronous message exchanges between the nodes.

There are several researches to implement the multi-rendezvous mechanism under the various restrictions [5, 9, 44]. Under the assumption that a tuple of synchronizing processes are never changed, these researches statically compose the special logical graphs connecting the synchronizing nodes to reduce the message exchanges.

In the future, we would like to implement an efficient multi-rendezvous mechanism which allows a tuple of the synchronizing nodes to be dynamically decided. We think it may be possible by using a broadcast mechanism in bus-connected networks. Design and development of an algorithm for such a multi-rendezvous mechanism, and implementation of the algorithm within the LOTOS compiler are our future work.

References

[1] Abe, K., Matsuura, T. and Taniguchi, K.: "An Implementation of Portable Lightweight Process Mechanism under BSD UNIX", *Journal of Information Processing Society of Japan*, Vol. 36, No. 2, pp.296-303 (1995) (in Japanese).

[2] Aho, A.V., Sethi, R. and Ullman, J. D.: Compilers–Principles, Techniques and Tools–, Addison-Wesley, p.796(1986).

[3] Amer, P. D. and New, D.: "Protocol Visualization in Estelle", *Computer Networks and ISDN Systems 25*, pp.741-760 (1993).

[4] Bochmann, G. v. and Gotzhein, R., Deriving protocol specifications from service specifications, *Proc. SIGCOMM'86*, pp.144-156(1986).

[5] Bochmann, G. v., Gao, Q. and Wu, C.: "On the Distributed Implementation of LOTOS", Proc. of the 2nd International Conference on Formal Description Techniques (Forte'89) (1989).

[6] 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.

[7] Chanson, S. T., Loureiro, A. A. F. and Vuong, S. T.: On tools supporting the use of formal description techniques in protocol development, *Computer Networks and ISDN Systems*(1993).

[8] Cheng, Z., Takahashi, K., Shiratori, N. and Noguchi, S.: An Automatic Implementation Method of Protocol Specifications in LOTOS, *IEICE Trans. Inf. & Syst. of Japan*, Vol. E75-D, No. 4(1992).

[9] Cheng, Z., Huang, T. and Shiratori, N.: "A New Distributed Algorithm for Implementation of LOTOS Multi-Rendezvous", Proc. of the 8th. IFIP Intl. Conf. on Formal Description Techniques (FORTE'94) (1994).

[10] Cooper, E. and P. Draves, R.: C Threads, *TR CMU-CS-88-154*, Department of Computer Science, Carnegie Melon University (1988).

[11] Curtis, B., Kellner, M. and Over, J. : Process Modeling, *Commun. ACM*, Vol. 35, No. 9, pp. 75 – 90 (1992).

[12] Deiters, W. and Gruhn, V.: Managing Software Processes in the Environment MELMAC, *ACM SIGSOFT*, Vol.15, No. 6(1990).

[13] Dubuis, E.: An Algorithm for Translating LOTOS Behavior Expressions into Automata and Ports, *Proc. of the 2nd Formal Description Techniques(FORTE'89)*, pp.163-177(1990).

[14] Ehrig, H. and Mahr, B. : Fundamentals of Algebraic Specification 1,

EATCS Monographs on Theoretical Computer Science, Vol. 6, Springer-Verlag(1985).

[15] P. H. J. van Eijk, C.A. Vissers and M. Diaz : "The Formal Description Technique LOTOS", North Holland, 1989.

[16] P. v. Eijk and H. Eertink : "Design of the LOTOSPHERE Symbolic LOTOS Simulator", Proc. of the Formal Description Technique III, North-Holland, pp.577-580, 1990.

[17] Van Eijk, P., Kremer, H. and Van Sinderen, M.: On the use of specification styles for automated protocol implementation from LOTOS to C, *Proc. of the 10th Int. Symp. on Protocol Specification, Testing, and Verification(PSTV-X)*, pp.157-168(1990).

[18] Gilbert, D.: Executable LOTOS: Using PARLOG to implement an FDT, *Proc. of the 7th Int. Symp. on Protocol Specification, Testing, and Verification(PSTV-VII)*, pp.281-294(1987).

[19] 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.

[20] R. Guillemot, M. Haj-Hussein and L. Logrippo : "Executing Large LOTOS Specifications", *Proc. of 8th IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification*, North Holland, pp.399-410, 1988.

[21] T. Higashino, H. Seki and K. Taniguchi : "Refinements of Algebraic Specifications to Functional Programs and Their Efficient Execution", J. IPS, Vol. 29, No. 8, 1988 (in Japanese).

[22] Higashino, T. and Taniguchi, K. : A System for the Refinements of Algebraic Specifications and their Efficient Executions, *Proc. of the 24th Hawaii Int. Conf. on System Sciences(HICCS-24)*, Vol. II, pp.186-195(1991).

[23] T. Higashino, R. Kato, 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).

[24] T. Higashino : "Service Specification and Its Protocol Specifications in LOTOS - A Survey for Synthesis and Execution -", IEICE Trans. Fundamentals, Vol. E75-A, No. 3, pp.330-338, 1992.

[25] Higashino, T., Bochmann, G. v., Li, X., Yasumoto, K. and Taniguchi, K. : A Test System for a Restricted Class of LOTOS Expressions with Data Parameters, *Proc. of the 5th IFIP Workshop on Protocol Test Systems*, pp.205-216(1992).

[26] Higashino, T., Okano, K., Imajo, H. and Taniguchi, K. : Deriving Protocol Specifications from Service Specifications in Extended FSM Models, *Proc. of the 13th IEEE Int. Conf. on Distributed Computing Systems*, pp.141-

148(1993).

[27] G. J. Holzmann : "Design and Validation of Computer Protocols", Prentice Hall Software Series, Prentice Hall, 1991.

[28] Huff, K. E. and Lessor, V. R.: A plan-based intelligent assistant that supports the software development process, *Proc. of the 3rd Software Engineering Symposium on Practical Software Development Environments, Software Eng.*, Notes 13, 5, pp.97-106(1989).

[29] IEEE. Threads Extension for Portable Operating Systems(Draft 6), February 1992, *P1003.4a/D6*(1992).

[30] Inoue, K., Seki, H., Taniguchi, K. and Kasami, T. : Compiling and Optimizing Methods for the Functional Language ASL/F, *Science of Computer Programming*, Vol. 7, No. 3, pp.297-312(1986).

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

[32] ISO : G-LOTOS : A Graphical Syntax for LOTOS, *ISO/IEC JTC1/SC21 N 3253*(1989).

[33] P. C. Kanellakis and S. A. Smolka : "CCS Expression, Finite State Processes, and Three Problems of Equivalence", Information and Computation, Vol. 86, pp.43-68, 1990.

[34] Kant, C., Higashino, T. and Bochmann, G. v. : Deriving Protocol Specifications from Service Specifications Written in LOTOS, *Proc. of the 12th Int. IEEE Phoenix Conference on Computers and Communications*, pp.310-318(1993).

[35] Kellner, M. : Software Process Modeling Example Problem, *Proc. of the 1st Int. Conf. on Software Process*, pp.176-186(1991).

[36] F. Khendek, G. von Bochmann and C. Kant : "New results on deriving protocol specifications from services specifications", *Proc. of the ACM SIGCOMM'89*, pp.136-145, 1989.

[37] Langerak, R.: Decomposition of functionality: A correctness-preserving LOTOS transformation, *Proc. of the 10th IFIP Int. Symp. Protocol Specification, Testing, Verification*, pp.203-218(1990).

[38] X. Li, K. Yasumoto, T. Higashino and K. Taniguchi : "A Test System for a Restricted Class of LOTOS Expressions", Technical Report of IPS Japan, 91-DPS-54-4, pp.25-32, 1992 (in Japanese)

[39] Matsuura. T, Nakamura, T., Higashino, T., Taniguchi, K. and Masuda. S : VTM : A Graph Editor for Large Trees, *Proc. of the IFIP 12th World Computer Congress*, Vol. I, pp.210-216(1992).

[40] T. de Miguel, T. Robles, J. Salvachua and A. Azcorra : "The SRTS experience: Using TOPO for LOTOS Design and Realization", Proc. of the Formal Description Techniques III, North-Holland, pp.383-394, 1990.

[41] Manas, J. A., Salvachia, J.: $\Lambda\beta$: a Virtual LOTOS Machine, *Proc. of the 4th Formal Description Techniques(FORTE'91)*, pp.445-460(1991).

[42] Milner, R.: A calculus of communicating systems, Springer Lecture Notes in Computer Science, Vol. 92, Springer-Verlag (1980).

[43] Mueller, F.: A Library Implementation of POSIX Threads under UNIX, *1993 Winter USENIX*(1993).

[44] Naik, K.: "Distributed Implementation of Multi-rendezvous in LOTOS Using the Orthogonal Communication Structure in Linda", Proc. of the 15th Int. Conf. on Distributed Computing Systems (ICDCS-15), pp. 518 – 525 (1995).

[45] Nakata, A., Higashino, T. and Taniguchi, K. : LOTOS Enhancement to Specify Time Constraints among Non-adjacent Actions using 1st-order Logic, *Proc. of the 6th Int. Conf. on Formal Description Techniques*(1993).

[46] Nakayama, T., Higashino, T. and Taniguchi, K.: Derivation of Software Process Description for each Developer from Whole Process Description written in LOTOS, *Technical Report of IEICE of Japan*, COMP 91-65(SS 91-22), pp.59 – 67 (1991) (in Japanese).

[47] Nomura, S., Hasegawa, T. and Takizuka, T. : A LOTOS Compiler and Process Synchronization Manager, *Proc. of the 10th Int. Symp. on Protocol Specification, Testing, and Verification (PSTV-X)*, pp.169-182(1990).

[48] K. Ohmaki, K. Takahashi and K. Futatsugi : "A LOTOS Simulator in OBJ", Proc. of the Formal Description Technique III, North-Holland, pp.535-538, 1990.

[49] N. Shiratori, H. Kaminaga, K. Takahashi and S. Noguchi : "A Verification Method for LOTOS Specifications and its Application", Proc. of the Ninth IFIP WG 6.1 Symposium on Protocol Specification, Testing and Verification, North Holland, pp.59-70, 1989.

[50] Shiroshima, T., Matsuura, T. and Taniguchi, K.: "Implementation and Evaluation of an Interactive Animation System", *Proc. of Computer Symposium of IPS Japan*, Vol. 95, No. 7 (1995) (in Japanese).

[51] Sun Microsystems: System Services Overview, pp.71-106(1988).

[52] 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).

[53] Turner, K. J. and McClenaghan, A.: "Visual Animation of LOTOS using SOLVE", *Proc. of the 7th Formal Description Techniques (FORTE'94)*, pp.

283-285 (1994).

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

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

[56] Yasumoto, K., Higashino, T. and Taniguchi, K. : "Execution of Protocol Specifications Written in LOTOS", *Technical Report of IEICE of Japan*, IN91-112, pp. 51 – 56 (1991) (in Japanese).

[57] Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K. : PROSPEX: A Graphical LOTOS Simulator for Protocol Specifications with N Nodes, *IEICE Trans. Commun.*, Vol.E75-B, No 10, pp.1015 – 1023(1992).

[58] Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K.: "Visualizing Dynamic Behavior of LOTOS Specifications", *I.C.S Research Report, 95-ICS-3*, Dept. of Information and Computer Sciences, Osaka University (1995).

[59] Yasumoto, K., Higashino, T., Abe, K., Matsuura, T. and Taniguchi, K.: A LOTOS Compiler Generating Multi-threaded Object Codes, *Proc. of the 8h Formal Description Techniques (FORTE'95)*, Chapman & Hall, pp. 271 – 286 (1995).

[60] Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K.: "Protocol Visualization using LOTOS Multi-Rendezvous Mechanism ", *Proc. of the IEEE 1995 Int. Conf. on Network Protocols (ICNP-95)*, pp. 118 – 125 (1995).

List of Figures

List of Tables