Implementation of Distributed Systems described with LOTOS Multi-rendezvous on Bus Topology Networks

Keiichi Yasumoto[†], Kazuhiro Gotoh^{††}, Hiroki Tatsumoto^{††}, Teruo Higashino^{††} and Kenichi Taniguchi^{††}*

[†] *Dept. of Information Processing and Management, Shiga Univ., 1-1-1 Banba, Hikone 522, JAPAN, E-mail: yasumoto@biwako.shiga-u.ac.jp*

^{††} *Dept. of Information and Computer Sciences, Osaka Univ., Toyonaka, Osaka 560, JAPAN, +81-6-850-6607 (Tel.), -6609 (Fax.)
E-mail: {tatumoto,higashino,taniguchi}@ics.es.osaka-u.ac.jp*

^{*} *Currently SONY Ltd., E-mail: gotoh@sm.sony.co.jp*

Abstract

In this paper, we propose an implementation method for specifications of distributed systems described in a subclass of LOTOS where operators such as choice and disabling can be used in combination with multi-rendezvous among remote processes. A LOTOS specification with the assignment of each process to a node is implemented as a set of executable codes which run on the corresponding nodes cooperating with each other by exchanging messages. The processes assigned to a node are transformed into a multi-threaded C code by our existing LOTOS compiler. Here, we focus on bus topology networks where broadcasted messages are received at all nodes in the same order, and propose a technique to implement multi-rendezvous among remote processes located on different nodes. We have also extended our LOTOS compiler to generate a set of C codes running on Ethernet. Some experimental results show that typical distributed systems can be described and implemented efficiently.

Keywords

LOTOS, multi-rendezvous, implementation, compiler, distributed systems

1 INTRODUCTION

The formal specification language LOTOS (ISO 1989) has advanced communication primitives such as multi-rendezvous as well as choice and interruption among concurrent processes. With those primitives, we can easily describe specifications of distributed systems such as mutual exclusion systems con-

sisting of cooperating multiple remote workstations. For the rapid prototyping and the performance evaluation, it is desirable to describe the distributed systems in abstract way using FDT, and to automatically generate efficient executable codes (object codes). For these purposes, it is desirable (1) that system specifications can be simply described (it depends on the class of description languages) and (2) that efficient object codes can be generated.

In general LOTOS specifications, we can use choice, interleaving and disabling operators in combination with synchronization operators among multiple processes. In those specifications, several mutually exclusive rendezvous may become executable among several combinations of processes simultaneously (ISO 1989). This causes a number of message exchanges for making a consensus among remote processes when we implement multi-rendezvous in distributed environments.

Cheng *et al.* (1994) and Naik (1995) have proposed algorithms for a subclass where the combination of synchronizing processes at each gate is always same but each process may participate in either of several rendezvous at different gates. However, since they assume general networks where broadcasted messages may be received in the different order at each node, they require many message exchanges dependent upon n in the worst case to make a consensus among processes (where n is the number of all nodes in a distributed system). It is not efficient to implement those algorithms on bus topology networks in spite of their property that the broadcast messages are received in the same order at all nodes. On the other hand, for describing system specifications simply, we should be able to use a general multi-rendezvous where the combination of synchronizing processes may dynamically change and each process does not know its synchronization peers in advance. Sisto *et al.* (1991) proposed the algorithm to deal with a general multi-rendezvous, which allocates a behavior expression of LOTOS to the subprocesses on a binary tree. However, this technique requires extra control processes between any two processes and the complex hierarchical communication structure among the processes.

In this paper, we focus on bus topology networks, and propose an efficient implementation method of a general case of LOTOS multi-rendezvous. In a subclass of LOTOS implemented in our method, we can specify multiple concurrent processes where each process should be assigned to a node. In a behavior expression of each process, I/O events including internal events, events synchronizing with other remote processes and process instantiations to other nodes can be specified. A distributed system is described as a set of those processes and the relation among them with LOTOS operators. Such a LOTOS specification is implemented as a set of object codes which run on the corresponding nodes cooperating with each other by exchanging messages. The processes assigned to a node are implemented as a multi-threaded object code by our LOTOS compiler proposed in (Yasumoto *et al.* 1995). In this paper, we propose a technique to implement operators specified among remote processes executed at different nodes. In our technique, each node selects

and executes the first enabled rendezvous independently from the contents and temporal ordering of the received messages under the assumption that broadcast messages are received in the same order at all nodes without failure.

In a general class of multi-rendezvous, the *executorial dependence relation* which denotes the relation among processes specified with LOTOS operators, may dynamically change due to process instantiations and terminations. To deal with such a case, each object code needs to know the current relation among remote processes. Therefore, we implement each object code so that it keeps information about a syntax tree of operators, specified among the processes as their executorial dependence relation. To keep the latest relation, when each process is invoked or terminated at a certain node, a message for invocation/termination is broadcasted with the location of the process in the current syntax tree so that all other object codes can update the information.

Although several LOTOS compilers have been developed (Dubuis 1990, Manas and Salvachia 1991, Nomura *et al.* 1990, Yasumoto *et al.* 1995), they can only generate object codes for a single node. So, we have extended our LOTOS compiler to generate a set of object codes running on Ethernet.

Some specifications of typical distributed systems such as mutual exclusion systems are provided for showing the expressiveness power of our subclass of LOTOS. In our experiments on Ethernet, it is shown that some example specifications can be implemented efficiently by our new compiler.

2 TARGET MODEL: CLASS AND ASSUMPTIONS

2.1 Class and assignment of processes to nodes

In our method, we describe a specification of a distributed system in a subclass of LOTOS with choice, interleaving, synchronization, disabling and enabling operators among multiple processes.

Let $M = \{M_1, \dots, M_n\}$ be the set of all nodes in a distributed system. Let L_j be a set of local events (including internal events) not synchronizing with processes in other nodes, S_j be a set of events synchronizing with processes in other nodes, and $A_j = L_j \cup S_j$ be a set of all events executed in M_j .

Let $P = \{p_1, p_2, \dots, p_m\}$ be a set of all processes in a LOTOS specification. We assign each process $p_i \in P$ to a node. Let $P(M_j) = \{p_{j,1}, \dots, p_{j,k}\} (\subseteq P)$ be a set of processes assigned to node M_j . A behavior expression of each $p_{j,i}$ is described with events in A_j , process instantiations of P and operators. The syntax of the LOTOS specifications is restricted to the class of Table 1.

In Table 1, we can specify one of the following three types of behavior expressions for each process $p_{j,i}$: (1) *global behavior expression (GB)* consisting of process instantiations of P and operators, (2) *local behavior expression (LB)* consisting of events in A_j and local process instantiations of $P(M_j)$, and (3) *hybrid behavior expression (HB)* which is an expression composed of two types

Table 1 Class of description

$HB ::= GB \mid LB \mid HB >> HB$
$GB ::= GB \parallel GB \mid GB \parallel GB \mid GB[[G]]GB \mid GB \parallel GB \mid GB[> GB$ $\mid GB >> GB \mid p(* node_list *) \mid p'$
$LB ::= stop \mid exit \mid Action; LB \mid LB \parallel LB \mid LB \parallel LB \mid LB[[G]]LB$ $\mid LB \parallel LB \mid LB[> LB \mid LB >> LB \mid p'$
$Action ::= gate \mid gate IOList \mid gate IOList [guard] \mid i$
$IOList ::= IO \mid IO IOList$
$IO ::= ?var : sort \mid !exp$

(Here, G denotes a list of gates whose events must synchronize between both operands of $[[G]]$. i is an internal event. p' is a process instantiation for the same node, and $p(*|node_list|*)$ for other nodes. Infinite behavior like $p := B \parallel p$ is prohibited)

Figure 1 Mutually exclusive multi-rendezvous

of expressions GB and LB . In HB , note that different types of expressions cannot be combined with operators except ' $>>$ ' (e.g. $a!3;exit \parallel P(*|node_1|*)$ cannot be described, but $Q \parallel P(*|node_1|*)$ where $Q := a!3;exit$ can do).

By specifying two executional dependence relations among remote processes and among local processes separately by GB and LB , respectively, the mechanism to control remote processes with calculating the events to be executed among them can be separated from the control mechanism of local processes executed within a node.

In each process instantiation in GB , $node_list$ denotes the nodes in which the specified process p may be invoked. For the purpose of load sharing, the process p can be executed in any node listed in $node_list$.

2.2 Problems to implement multi-rendezvous and assumptions for implementation

Even if only synchronization operators are specified among processes, several mutually exclusive rendezvous may be executable simultaneously when each process includes choices of different events (see Fig.1).

Figure 2 Basic policy to implement a distributed system

In order to implement the multi-rendezvous mechanism correctly, remote processes on different nodes need to exchange messages with each other for making a consensus to select a rendezvous. If we implement the mechanism on a general network where the broadcasted messages may be received in the different order at each node, we cannot take the strategy of selecting first enabled rendezvous from the temporal order of received messages.

So, on general networks, each process must decide a rendezvous after receiving the messages from all its synchronization peers. It results in $O(n^2)$ (Bochmann *et al.* 1989), $O(n\sqrt{n})$ (Cheng *et al.* 1994) or $O(n \log n)$ (Naik 1995) of message exchanges for each selection of a rendezvous where n is the number of nodes in the distributed system. Fig.1 is among the worst cases where any two exclusive rendezvous share one process.

To avoid the above problems and to implement a general case of multi-rendezvous efficiently, we focus on bus topology networks where the broadcasted messages are received in the same order at every node.

Assumptions for the target network

We assume the following properties for bus topology networks.

- (1) each message is received at all nodes in a distributed system by a broadcast
- (2) all messages are received at each node without failure (i.e without losses and duplications) and stored in its message buffer
- (3) broadcasted messages are received in the same order at all nodes

3 ALGORITHM

3.1 Basic policy of implementation

In our implementation method, we generate a set of object codes for the corresponding nodes in a distributed system from a given LOTOS specification as described in the class of Table 1. The generated object codes implement the given LOTOS specification, cooperating with each other (Fig.2).

Table 2 An example of LOTOS specification

```

spec. := ( P (*|node1|*) [] Q (*|node2|*) || R (*|node3|*) )
        [> ( S (*|node1|*) ||| T (*|node4|*) )
where P := (a!3;exit) >> ( P1 (*|node1, node2|*) || P2 (*|node3|*) )
        Q := b;Q
        R := a?x:int[x>0]; R [] b;R
        S := (S1 || S2) [] S3
        T := d;stop
(Here, the declarations of processes P1, P2, S1, S2 and S3 are omitted)

```

For example, four object codes for *node₁*, *node₂*, *node₃* and *node₄* are generated from the LOTOS specification in Table 2, and those object codes can execute the processes {P,S,P1}, {Q,P1}, {R,P2} and {T}, respectively. Since P1 may be executed on either *node₁* or *node₂*, both nodes have the code for executing P1. First, those object codes execute the processes in the initial behavior expression. In the example of Table 2, when the object code of *node₁* is executed, processes P and S are invoked.

The initial executional dependence relation among processes corresponds to the area within the dotted rectangle in Fig.3(i). In the relation, two exclusive rendezvous *a!3* and *b* can be executed with the combinations (P, R) and (Q, R), respectively. Several disabling events may be also executed in S and T.

The executional dependence relation among processes changes dynamically when new processes are invoked and/or processes are terminated. In the example of Table 2, when a rendezvous *a!3* of (P, R) is executed, process Q must be terminated and new processes P1 and P2 will be invoked. After that, the dependence changes from (i) to (ii) in Fig.3.

Therefore, in the proposed technique, we make each node hold the newest information about the executional dependence relation among all running processes of all nodes so that the temporal ordering of events among the processes are implemented based on the relation. In each object code, we allocate a working space called *control area* which has the same structure as the syntax tree of the operators specified among processes in order to memorize the initial executional dependence relation. When each process is invoked or terminated, the process broadcasts the information to all nodes so that the control area can be updated at each node to keep the newest relation.

Each process calculates its *synchronization events* which need to synchronize with other processes, independently based on the technique in (Yasumoto *et al.* 1995) and broadcasts a request message for the events. Then the process selects and executes a rendezvous which has satisfied the synchronization conditions* earlier than other rendezvous from the current executional dependence relation and the temporal order of the received messages in its buffer.

*In each rendezvous, I/O values of all of its participants must be unified (ISO 1989).

Figure 3 Executional dependence relation and its dynamic change

Figure 4 Environment of each node

3.2 Management of control among processes

We assume that the broadcasted messages from all nodes are received and stored in the message buffer of each node automatically and that each node has a sufficient buffer. Each process calculates a set of its synchronization events $\{a_1, \dots, a_k\}$, and broadcasts a request message for the set (denoted by $req(\{a_1, \dots, a_k\})$). In an object code of each node, we implement an additional process called *message handler* to select a rendezvous from the current executional dependence relation and both the contents and temporal ordering of the messages stored in the buffer. Each process executes the selected event after the message handler permits its execution. Fig.4 shows the environment of $node_1$ in the example of Table 2.

Here, it is important (1) that choice and disabling operators among processes should be implemented correctly, (2) that one of the exclusive ren-

deztous should be selected under a consensus of all nodes, and (3) that the dynamic change of the executional dependence relation should be managed correctly among all nodes.

The above criterion (1) can be implemented fairly with the following policy: if a process p wants to execute its events, p broadcasts a request message for those events with the process name ' p '. When the message handler of each node processes the message, it examines the path (called *control path*) from p 's position (leaf) to the root node in the control area and checks whether different sides of behavior expressions have been selected and/or an interruption has occurred. Only when the handler detects that every different side of expressions has not been selected at each node, it accepts the message, and updates the control path so that p 's side has been selected. After that, p executes one of those events at random. In this policy, the process which broadcasted a request message earlier than others is selected and executed.

In the example of Table 2, suppose that the request messages of events $req(\{d\})$ and $req(\{a!3\})$ from processes T and P are broadcasted in this order. The message handler of each node stores the information that interruption occurred in the sub-area corresponding to $[>]$ operator when it processes $req(\{d\})$. When the later message $req(\{a!3\})$ is processed, the message handler detects the occurrence of interruption on the control path of P. Then, $req(\{a!3\})$ is rejected and process P is terminated.

The criterion (2) is implemented as follows: when each message handler processes a request message m for synchronization events, the handler stores the information about the synchronization events in the area corresponding to $[|G|]$ operator which is the nearest to the leaf in the control path of m 's sender. If the handler finds appropriate synchronization peers at the area, it applies the same operation to the upper areas of $[|G|]$ operators with new event values after matching synchronization conditions. If there is no upper $[|G|]$ in the control path, then the handler permits the events to be executed.

We implement the criterion (3) as follows: when a process instantiation becomes executable in a process at a node, the message handler of the node calculates the control path of the process instantiation and broadcasts a request message for the process instantiation with its control path. When the message handler at each node processes the message, it extends its control area with the control path in the message and it also invokes the process if it is requested to be invoked at the node. Once a process is selected to execute an event, the subareas corresponding to choice and disabling operators in the control area may become unnecessary. Such unnecessary areas are removed when the messages of all processes that were not selected have been processed.

We will explain more about the criteria (2) and (3). Suppose that the request messages $req(\{a!3\})$, $req(\{a?x : int[x > 0], b\})$ and $req(\{b\})$ are broadcasted from P, R and Q in this order (see Fig.4). When each message handler processes $req(\{a!3\})$ from P, it registers the information $a!3$ to the area corresponding to $||$ on the control path of P. When the next message

$req(\{a?x : int[x > 0], b\})$ from R is processed, a rendezvous between P and R satisfies the synchronization conditions with $a!3$ and both P and R are not disabled on their control paths. So $a!3$ is selected and executed between P and R after updating the control paths of P and R (the side of P is stored to the area corresponding to $\square$). After that, when $req(\{b\})$ from Q is processed, Q detects that the different side has been selected from the information stored at the area of $\square$ on its control path. So, Q is terminated at $node_2$. On the other hand, if the request messages from P, Q and R are broadcasted in this order, R's request message $req(\{a?x : int[x > 0], b\})$ enables two exclusive rendezvous $a!3$ and b simultaneously. To cope with this problem, we adopt the technique based on the random number introduced in (Cheng *et al.* 1994, Naik 1995). Here, we attach a random integer to each event in the request messages for calculating the maximum value of each rendezvous so that all nodes can uniquely select the rendezvous with the largest number.

When $a!3$ is executed in P, the behavior expression of P changes to $P1 \parallel P2$ and the request messages for process instantiations $inst(P1)$ and $inst(P2)$ are broadcasted. Each message includes the process name, node names and the control path. If several nodes are specified like $P1(*|node_1, node_2|*)$, the process is invoked at the node whose load is the lowest. The information about the load of each node is attached to each request message so that all nodes can keep the information to find the lowest node.

Treatment of local processing in each process

In the class of Table 1, since each process may include multiple concurrent sub-processes executed in its node. It may cause the messages for several exclusive events to be broadcasted separately. For example, in process $P := (a; exit \square b; exit) \parallel (...; a; ... \parallel ...; b; ...)$, two messages for exclusive events a and b may be broadcasted at the different time. To deal with such a case correctly, we use a message for replacing an issued request message to a new one. If we need to broadcast a request message of alternative event b after broadcasting the request message $req(\{a\})$, we can broadcast $replace(req(\{a\}), req(\{a, b\}))$ to change the issued request message. Since a message handler processes messages sequentially according to their temporal ordering, the replaced message may become obsolete when a rendezvous has been selected before the message is processed. In that case, the message will be ignored.

3.3 Correctness and efficiency of the algorithm

Here, we briefly explain the correctness and the efficiency of our algorithm.

Correctness

The executable events among processes depend on the current executional dependence relation among the processes and *active events* which want to be executed in each process. If rendezvous, choice and/or disabling operators are specified among processes, the executional dependence relation may change

after an event (or a rendezvous) has been executed. So, if all nodes can keep the latest information about the relation, and if each node can calculate the same event (rendezvous) of a process (a combination of processes) from the relation and active events in each process, the algorithm will work correctly.

In the algorithm, if choice and/or disabling operators are specified among processes, the process which has sent the earliest request message among the processes is selected. For synchronization operators, the combination of the processes whose messages have been sent earlier than others is selected. The algorithm assumes that the order and the contents of the received messages are the same among all nodes as explained in Sec.2.2 and all nodes work based on the same algorithm. This fact shows that each node is capable of knowing which processes are selected/invoked, and therefore of keeping the current executional dependence relation at any time, with sequentially processing messages in its local buffer. Therefore, the algorithm works correctly.

Here, we mention a bit about the fairness of our algorithm. If the performances of nodes differ greatly, some rendezvous may be always selected among the nodes with good performances and other alternative rendezvous containing 'slow' nodes may not be selected infinitely. It is because the algorithm always selects the first enabled rendezvous from the temporal ordering of received messages. In that case, the fairness in the nondeterministic selection may not be preserved in the algorithm. However, if a message makes several exclusive rendezvous executable at the same time (see Sec.3.2), each of them can be selected based on random numbers.

Efficiency compared to other algorithms

Since each process broadcasts a request message for its active events at each stage, at most n broadcast messages are sent as a total with our algorithm (here, n is the number of all processes in the distributed system). Therefore, if our algorithm is implemented with the unicast mechanism, its message complexity will be $O(n^2)$, which may seem to be worse than other algorithms.

In the previously proposed algorithms (Bochmann *et al.* 1989, Cheng *et al.* 1994, Naik 1995), the temporal ordering of the received messages may be different because they assume general networks. This fact forces all processes to make a consensus for a rendezvous after receiving all messages from related processes if several exclusive rendezvous may exist. On the other hand, with our algorithm we can select the first enabled rendezvous owing to the property of bus topology networks. Accordingly, for selecting a rendezvous consisting of n_k participants (usually n_k is much less than n), our algorithm requires only n_k broadcast messages even in the worst case as shown in Fig.1. Although we have to consider there may be the messages broadcasted for other unselected rendezvous, most of them can be used for calculating the subsequent rendezvous. So, the average number of the required messages for a rendezvous approximately results in n_k . In addition, since the cost of a broadcast is almost same as that of a unicast on bus topology networks, the proposed algorithm may work much faster than the previous ones which assume general networks.

4 APPLICATION

This section shows some typical distributed systems which can be described simply within the class defined in Table 1.

Example 1. Duplication system

Let us suppose that a service is duplicated for high reliability. In the case that each service process fails rarely, it is popular to make several processes carry out the same task and to select their majority by getting the results from those processes.

Here, we describe a request for each task as $req?task$, and an output of the result after processing the task as $line!ProcessTask(task)$ in the following process $S[req, line]$ ($ProcessTask(task)$ is an abstract data type function which calculates the result from $task$).

$S[req, line] := req?task; line!ProcessTask(task); S[req, line]$

Next, we describe another process called *Checker* which gets the results from duplicated processes and outputs the correct result based on the majority or detects an error. *Checker* receives three results as x , y and z from three processes, respectively with multi-rendezvous. If more than two of x , y and z are the same ($Correct(x, y, z)$), *Checker* outputs their majority ($out!Major(x, y, z)$) and repeats the procedure. Otherwise, it executes the event er to stop all processes. *Checker* can be described as follows:

$Checker[ln1, ln2, ln3, out, er] :=$
 $(ln1?x; exit \parallel ln2?y; exit \parallel ln3?z; exit)$
 $>> ([Correct(x, y, z)] -> out!Major(x, y, z); Checker[ln1, ln2, ln3, out, er])$
 $\square [not\ Correct(x, y, z)] -> er; stop)$

We invoke three duplicated processes S in parallel on node1, 2 and 3, which send the results to *Checker* via three gates $ln1$, $ln2$ and $ln3$, respectively. The whole system can be described as follows:

$((S[req, ln1](\ast|node_1|\ast) \parallel [req]) S[req, ln2](\ast|node_2|\ast) \parallel [req]) S[req, ln3](\ast|node_3|\ast))$
 $[> Error[er](\ast|node_4|\ast)]$
 $\parallel [ln1, ln2, ln3, er] Checker[ln1, ln2, ln3, out, er](\ast|node_4|\ast)$
 where $Error[er] := er; stop$

Example 2. Client server system

In client server systems such as VOD (video on demand) systems, a series of data streams is transferred successively between client and server systems. With the class in Table 1, we can describe such a system.

If both video and audio data must be transmitted synchronously from server machines (SV and SA) to a client machine (C) and we would like to make

several nodes share their load to execute a server process, such a client server system can be described as follows:

$SV(*|node_1, node_2, node_3|*) \parallel [m] \ SA(*|node_1, node_2|*) \parallel [m] \ C(*|node_4|*)$
(Here, m denotes a gate for exchanging video and audio data)

Since some nodes can execute server processes SV and SA in our model, each server process is invoked at the node with the lowest load.

Example 3. Mutual exclusion system

In designing a distributed system, frequently we treat the mutual exclusion such that a resource placed at a node can be used by at most k user processes simultaneously.

If each user process uses the resource **A** after $a!lock$ and releases **A** before $a!unlock$, the restriction for **A** can be described as follows:

$R[a](usr, max_A) :=$
 $([usr < max_A] \rightarrow a!lock; R[a](usr + 1, max_A)) \parallel a!unlock; R[a](usr - 1, max_A)$
(max_A denotes the number of user processes which can use the resource simultaneously)

If processes $P_1, \dots, P_n$ using resource **A** can run independently of each other except the access to **A**, the whole system can be described as follows:

$(P_1[a](*|node_1|*) \parallel P_2[a](*|node_2|*) \parallel \dots \parallel P_n[a](*|node_n|*))$
 $\parallel [a] \ R[a](0, max_A)(*|node_1|*)$

In the above specification, events $a!lock$ and $a!unlock$ in process R synchronize to either of $P_1, \dots, P_n$ and increase and decrease the variable usr , respectively. If $usr + 1$ exceeds max_A , only $a!unlock$ can be executed. In that case, processes which want to use resource **A**, will wait until **A** is released with $a!unlock$.

We can easily extend the above system for treating several resources, say, **A**, **B** and **C**, only by adding processes for restricting those resources as follows:

$(P_1[a, b, c] \parallel P_2[a, b, c] \parallel \dots \parallel P_n[a, b, c])$
 $\parallel [a, b, c] \ (R[a](0, max_A) \parallel R[b](0, max_B) \parallel R[c](0, max_C))$
(Here, the node names for processes are omitted)

5 IMPLEMENTATION AND ITS EVALUATION

5.1 LOTOS compiler and implementation on Ethernet

Each local process whose behavior expression is described as an LB in Table 1 is implemented as a multi-threaded object code by our LOTOS compiler. The compiler implements each process containing concurrent subprocesses with our portable multi-thread mechanism PTL (Abe *et al.* 1995) as follows:
(1) decomposing each behavior expression into some sequentially executable

subbehavior expressions called *runtime units*, (2) mapping each runtime unit to a thread, and (3) constructing a shared memory (called *local control area*).

To implement a wide class of behavior expressions a local control area is used to implement the executional dependence relation among threads with a similar technique explained in Sec.3.2. Since each area in the local control area may be accessed by multiple concurrent threads simultaneously, the mutual exclusion mechanism in PTL is used. The above techniques enabled us to obtain a higher performance than other compilers in (Dubuis 1990, Manas and Salvachia 1991). The details are in (Yasumoto *et al.* 1995).

We have extended the compiler to generate a set of object codes which run on the corresponding nodes on Ethernet based on the algorithm in Sec.3.

Implementation of the algorithm on Ethernet

In popular LAN environments, Ethernet is widely used. Unfortunately, Ethernet does not satisfy the assumptions in Sec.2.2 partly. So, first we have examined to what extent Ethernet satisfies the assumptions. Here, we used UDP/IP on several UNIX workstations (BSD-OS 2.1 on Pentium 90 to Pentium Pro 200MHz) connected to fast ethernet network (100BASE-TX).

Through the experiments, we recognized (i) that a few percent of messages are lost when the frequency of message exchanges are high (e.g. when over 3000 messages a second), and (ii) that some messages are reordered on some nodes when several messages are broadcasted in a short time period.

The reason for (i) is that the message buffer on a slow machine overflows due to the shortage of the processing time for messages. (ii) is caused by the loop back mechanism of the operating system by which the messages sent from a node is stored to its message buffer directly.

For the above problems, we can use the techniques proposed in (Melliard *et al.* 1990, Luan and Gligor 1990) so that the assumptions hold on Ethernet. However, for efficiency, here we adopt to place a proxy server which broadcasts messages forwarded from all nodes. Each node does not broadcast its messages directly but asks the server to broadcast its messages. By placing the server at an independent node, all nodes in a distributed system can receive the messages in the same order. By adding sequential numbers to subsequent messages, each node can easily detect message loss and ask for retransmission from the server. Our current implementation includes such a proxy server.

5.2 Evaluation

To evaluate our implementation method, we have examined the efficiency of the object codes generated from several specifications of distributed systems.

In the experiments, we have measured the number of events synchronizing

Table 3 Performance of simple rendezvous

$k \backslash n$	$n=2$	$n=3$	$n=4$	$n=5$
$k=1$	993	818	647	571
$k=2$	663	534	426	367
$k=3$	481	370	319	257
$k=4$	519	290	237	190
$k=5$	428	225	184	139

Table 4 Performance of a mutual exclusion system

$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
680	688	586	592	598

Table 5 Efficiency in abracadabra protocol (KBytes/sec.)

packet size	$n=1$	$n=2$	$n=4$
256 bytes	8.7	7.7	6.0
512 bytes	18.7	16.1	12.0
1024 bytes	35.0	31.5	23.2

among remote processes executed in a time unit (events/second). The object codes were executed on the same environment in Sec.5.1.

Basic performance

First, we have executed object codes generated from the following specification where each event synchronizes among n remote processes repeatedly. The experimental result is shown in Table 3.

$P(*|node_1|*) || P(*|node_2|*) || \dots || P(*|node_n|*)$ where $P := a_1; P[]a_2; P[] \dots []a_k; P$

According to Table 3, the performance depends on both the numbers of processes and alternative events (n and k). The latter is due to the cost of selecting a rendezvous of several exclusive ones.

Performance of more complicated distributed systems

We have examined the efficiency for more complicated distributed systems.

First, we have measured the performance in a mutual exclusion system where n remote concurrent processes use a resource exclusively from each other as explained in Sec.4. We show in Table 4 the experimental results with several numbers for n . Table 4 shows that the performance does not depend on the number of user's processes. This is owing to the property of our algorithm that the request messages for the unselected rendezvous are used for selecting the subsequent rendezvous unless their sender processes are disabled.

Finally, we have examined the efficiency of a file transfer system based on a specification of abracadabra protocol (ISO/IEC/TR 10167 1991). The specification describes the behavior of a node where a file is transferred (received) to (from) other node depending on the request of the service user on the node. In the protocol, the file is divided into multiple data packets and the packets are transmitted via unreliable medium. The specification includes several concurrent subprocesses synchronizing with each other. Here, we extended the specification so that a sender node (*Sndr*) distributes a file (the size is

1 Mbytes) to n receiver nodes (*Recv*) synchronizing with a process *Medium* which emulates the communication medium.

$$\text{Medium}[m_1, m_2](\ast|\text{node}_1|\ast) \parallel [m_1, m_2] \\ (\text{Sndr}[a_1, m_1](\ast|\text{node}_2|\ast) \parallel \parallel \text{Recv}[a_2, m_2](\ast|\text{node}_3|\ast)) \dots \parallel \parallel \text{Recv}[a_n, m_n](\ast|\text{node}_{n+2}|\ast))$$

In Table 5, we show the result with the size of a data packet from 256 to 1024 bytes and n from 1 to 4. In Table 5, if the packet size is small (256 bytes), the transmission rate is lowering loosely as the number of recipients is growing. In contrast, the rate is lowering drastically if the size is large (1024 bytes). It may be caused by the overhead of local processing for messages.

6 CONCLUDING REMARKS

In this paper, we have proposed a technique to implement a specification with ‘general’ multi-rendezvous among multiple remote processes as a set of object codes which can be executed at the multiple remote nodes on bus topology networks. In our implementation method, a distributed system is described as a set of processes representing the individual behaviors of nodes. It might be useful to integrate the technique in (Kant *et al.* 1996) which automatically derives protocol entities’ specifications from a service specification (i.e. the whole behavior of a distributed system).

Although there is an opinion that the efficient implementations are not necessary for specification languages like LOTOS, we believe that the efficient implementations are important for rapid prototyping and other purposes. An application is the visualization of the dynamic behavior of distributed systems. With the visualization technique in (Yasumoto, Higashino *et al.* 1995), we can monitor the dynamic behavior of remote processes in a distributed system as the graphical animation in realtime. It may be useful for, say, the education of distributed systems in LOTOS. One of future work is to extend the compiler to deal with a time extended LOTOS (e.g. E-LOTOS).

REFERENCES

- Abe, K., Matsuura, T. and Taniguchi, K.: “An Implementation of Portable Lightweight Process Mechanism under BSD UNIX”, *J. of Information Processing Society of Japan*, Vol. 36, No. 2, pp. 296 – 303 (1995) (in Japanese).
- Bochmann, G. v., Gao, Q. and Wu, C.: “On the Distributed Implementation of LOTOS”, *Proc. 2nd Int. Conf. on Formal Description Techniques*, pp. 133 – 146 (1989).
- Cheng, Z., Huang, T., and Shiratori, N.: “A New Distributed Algorithm for Implementation of LOTOS Multi-Rendezvous”, *Proc. 7th Int. Conf. on Formal Description Techniques*, pp. 493 – 504 (1994).
- Dubuis, E.: “An Algorithm for Translating LOTOS Behavior Expressions into Au-

- tomata and Ports", *Proc. 2nd Int. Conf. on Formal Description Techniques*, pp. 163 – 177 (1990).
- ISO : "Information Processing System, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", *IS 8807* (1989).
- ISO/IEC/TR 10167: "Information Technology – Open Systems Interconnection – Guidelines for the application of Estelle, LOTOS and SDL" (1991).
- Kant, C., Higashino, T. and Bochmann, G.v.: "Deriving protocol specifications from service specifications written in LOTOS", *Distributed Computing*, Vol. 10, No.1, pp. 29 – 47 (1996).
- Manas, J. A., Salvachia, J.: " $\Lambda\beta$: a Virtual LOTOS Machine", *Proc. 4th Int. Conf. on Formal Description Techniques*, pp.445 – 460(1991).
- Melliari, P.M., Moser, L.E. and Agrawara, V.: "Broadcast Protocols for Distributed Systems", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 17 – 25 (1990).
- Naik, K.: "Distributed Implementation of Multi-rendezvous in LOTOS Using the Orthogonal Communication Structure in Linda", *Proc. 15th Int. Conf. on Distributed Computing Systems*, pp. 518 – 525 (1995).
- Luan, S. and Gligor, V.D. : "A Fault Tolerant Protocol for Atomic Broadcast", *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, No. 3, pp. 271 – 285 (1990).
- Nomura, S., Hasegawa, T. and Takizuka, T. : "A LOTOS Compiler and Process Synchronization Manager", *Proc. 10th Int. Symp. on Protocol Specification, Testing, and Verification*, pp. 169 – 182 (1990).
- Sisto, R., Ciminiera, L. and Valenzano, A.: "A Protocol for Multi-rendezvous of LOTOS Processes", *IEEE Trans. on Computer*, Vol. 40, No. 1, pp. 437 – 447 (1991).
- Yasumoto, K., Higashino, T., Abe, K., Matsuura, T. and Taniguchi, K.: "A LOTOS Compiler Generating Multi-threaded Object Codes", *Proc. 8th Int. Conf. on Formal Description Techniques*, pp. 271 – 286 (1995).
- Yasumoto, K., Higashino, T., Matsuura, T. and Taniguchi, K.: "Protocol Visualization using LOTOS Multi-Rendezvous Mechanism", *Proc. IEEE 1995 Int. Conf. on Network Protocols*, pp. 118 – 125 (1995).

7 BIOGRAPHY

Keiichi Yasumoto is an Assistant Professor in the Department of Information Management and Processing, Shiga University, Japan. Since April 1997, he is working as a visiting researcher at the University of Montreal, Canada. Kazuhiro Gotoh finished his Master course at the Department of Information and Computer Sciences, Osaka University, Japan in 1997 and is currently working in SONY Ltd. Hiroki Tatsumoto is a Master student in the same department. Teruo Higashino and Kenichi Taniguchi are an Associate Professor and a Professor in the same department, respectively. Their research interests include formal description techniques for communication protocols and distributed systems, and software engineering.