# Protocol Visualization using LOTOS Multi-Rendezvous Mechanism

Keiichi Yasumoto† Teruo Higashino†† Toshio Matsuura††† Kenichi Taniguchi††

† Department of Information Processing and Management, Shiga University, Hikone, Shiga 522, Japan
†† Department of Information and Computer Sciences, Osaka University, Toyonaka, Osaka 560, Japan
††† Faculty of Human Life Science, Osaka City University, Sumiyoshi-ku, Osaka 558, Japan

## Abstract

*In this paper, we propose a method for visualizing LOTOS specifications using multi-rendezvous mechanism. For visualization, we have extended LOTOS by introducing some primitive animation events. Using the extended LOTOS, we describe a visualization scenario for events which we would like to visualize their execution. Then, we execute the original specification and its visualization scenario in parallel under LOTOS multi-rendezvous mechanism so that the corresponding animation is activated when each event is executed. The pair of the original specification and its visualization scenario is converted into the multi-threaded object code using our LOTOS compiler. In our visualization method, we can specify the visualization scenario without modifying the original specification, and we can derive an object code which animates the original specification in real time. We have tried to visualize a LOTOS specification of "Dijkstra's dining philosophers", and evaluated the usefulness of our approach.*

## 1 Introduction

Protocol visualization is useful in facilitating to understand dynamic behavior of communication protocols. Recently, the researches for facilitating to understand and/or design the formal specifications of communication protocols using visualization techniques have been studied [2, 5].

A specification of a communication protocol (*original specification*) can be visualized by displaying an appropriate animation (which is described in advance) when each event is executed. In such a visualization, it is desirable that we can describe the scenario for visualization (*visualization scenario*) independently of the original specification without modifying it, and that we can change the animations dynamically depending on the timing of each event occurrence or the values of each input/output. To visualize the behavior of concurrent systems in real time, it is also desirable that we can execute the visualized specifications as fast as possible. In order to visualize complex behavior, the language for describing animations should be able to specify alternative and parallel execution among several animations.

For these purposes, we have extended LOTOS (which has been standardized within ISO)[3] to describe the visualization scenarios.

First, we describe a visualization scenario specifying animations which we would like to display when particular events in the original specification are executed. In the extended LOTOS, we can use primitive operations for animations such as registration, indication, movement and elimination of animation objects. Each of them is described as an event of LOTOS. Various animations can be described by combining these primitives and LOTOS operators such as choice, parallel, interruption and so on. Then, we get the *visualized specification* by combining the original specification and its visualization scenario to be executed in parallel under the multi-rendezvous mechanism. In the visualized specification, each event in the original specification synchronizes with the activation of its corresponding animation in the visualization scenario.

The multi-rendezvous mechanism of LOTOS enables the dynamic data exchange among multiple concurrent processes [3]. Since the data values of inputs/outputs in each event can be exchanged between the original specification and the visualization scenario by using the mechanism, we can select one of alternative execution paths in the scenario and/or change animations depending on the exchanged values.

For evaluating the usefulness of our approach, we have implemented a system to execute the visualized LOTOS specifications by extending our LOTOS compiler [8] so that we can deal with animations. To derive fast object codes, the compiler generates a multi-threaded object code from a visualized specification, where the multiple concurrent processes are mapped to the threads and the communication among them are implemented using the shared data. The generated codes use our Interactive Animation Server [4] to display animations. It receives the current values of the attributes for each animation object (such as location, color, priority to others, and so on), and reflects them on the graphical window at a fixed time interval (such as 30 frames/sec). In the generated object codes, each animation is mapped to a thread. In each thread, the attribute values of the animation object at



0-8186-7216-1/95 $04.00 

the next frame are calculated and sent to the Animation Server. Since the multi-thread mechanism takes a low overhead in context switching among a number of concurrent threads, the multiple animation objects can move smoothly in real time.

In the following Sec. 2, a method for visualizing LOTOS specifications is explained. Sec. 3 describes the facilities of the system executing the visualized specifications. In Sec. 4, we try to visualize a LOTOS specification of Dijkstra's philosophers. Some experimental results are shown in Sec. 5. In the final Sec. 6, we conclude the paper with a discussion of related work.

## 2 Our visualization method

A LOTOS specification is described as a set of processes. We can use the operators shown in Table 1 to specify the temporal order of events in a behavior expression of each process. For executing the events sequentially, we combine those events with the action prefix operator ";". We can use the choice operator "[]" for alternative execution between processes. Each process combined by the interleaving operator "|||" can execute the events in parallel independently of other processes. We can also use the disabling operator "[>" to stop the all running processes when an interruption process starts running.

### 2.1 How to describe scenarios

LOTOS has a multi-rendezvous mechanism[3] 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.

Table 1: LOTOS operators

| Contents | Operators |
|---|---|
| 1. action prefix | $\alpha; B$ |
| 2. choice | $B_1 [] B_2$ |
| 3. interleaving | $B_1 \| \| \| B_2$ |
| 4. synchronization | $B_1 \|[G]\| B_2$ |
| 5. enabling | $B_1 >> B_2$ |
| 6. disabling | $B_1 [> B_2$ |

(here, $\alpha$ represents an event, and for $B, B_1$ and $B_2$, we can describe an action-prefixed sequence or any sequence connected by the above operators)

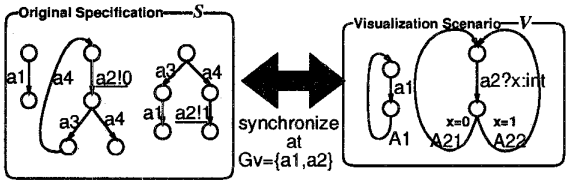


Figure 1: Our visualization method

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. 1):

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

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[3]. 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[3] 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.

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

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.

## 2.2 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.

Table 2: 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 |
| ... | ... |

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 Sec.2.3). 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.

AE?fkid : cast_t[fkid = CreateCast("fork.xbm")]

We show the part of animation primitives in Table 2. Using the operators in Table 1, we can describe various animations such that the multiple Casts are moving in parallel.

## 2.3 Structural visualization

In LOTOS, it is recommended that the specifications are described in the resource oriented and/or constraint oriented styles[6]. 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:

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

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

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

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. [7].

## 3 Executing visualized specifications

We use our LOTOS compiler [8] to convert the visualized specifications into the executable object codes.

### 3.1 LOTOS compiler

In order to execute visualized LOTOS specifications efficiently, it is important how to implement the specifications containing multiple concurrent processes. One of efficient implementation techniques is to use multi-thread mechanisms.

The multi-thread mechanism can handle concurrent threads (light-weight processes) efficiently within a process. There are several implementations such as Light Weight Process (LWP) mechanism within Sun OS, C Threads of Carnegie Mellon University for Mach OS, and so on. However these thread libraries depend on a particular architecture, for example, LWP depends on Sun OS. So, we have implemented Portable Thread Library (PTL) [1]. The characteristics of our PTL are as follows:

(1) describing all library codes only in C language for portability.

(2) also preparing architecture dependent codes and using them instead of the portable code if it improves the performance.

(3) providing standard application interface for general purposes (POSIX 1003.4a).

We have implemented LOTOS specifications under our multi-thread mechanism PTL as follows:

- decompose a behavior expression into some sequentially executed event sequences called *run-time units*.
- map each run-time unit to a thread.
- construct a shared memory which keeps the temporal ordering of events among all run-time units.

To implement general behavior expressions where the multiple operators are specified hierarchically, (1) the nodes in the shared memory corresponding to the operators should be referred hierarchically from each run-time unit, and (2) each run-time unit should have the ordered information how it connects to each operator. This information can be calculated statically in our LOTOS compiler. Under multi-thread environment, a variable may be accessed by multiple threads simultaneously. Using the mutual exclusion mechanism in PTL, multiple run-time units can avoid accessing the same place in the shared memory. Using those techniques, we can obtain a higher performance than the compilers which do not use multi-thread mechanisms. The details of the compiler are described in Ref. [8].

We have also extended the LOTOS compiler to treat the animations. We have used our Interactive Animation Server[4] to display the animations in the generated object codes. Since most of the animation primitives defined in Table 2 are equivalent to the instructions on Animation Server, the LOTOS compiler only replaces the animation primitives to the subroutine calls in C language.

### 3.2 Mechanism for real-time animation

Our Animation Server[4] 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. 2).

[Behavior of a module for moving each Cast]

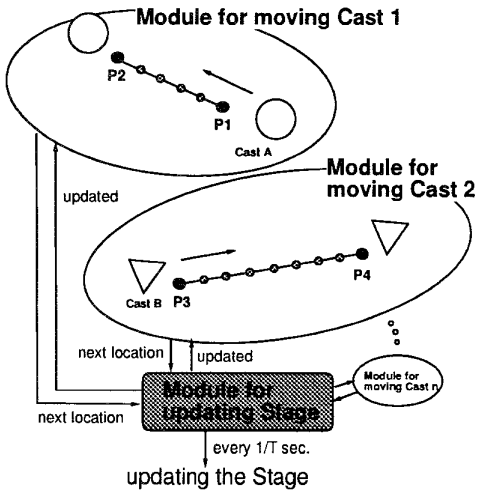


Figure 2: Mechanism for real-time animations

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.

## 4 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, ···, ph5 and fk1, ···, fk5 for distinguishing five philosophers and five forks, respectively (here, fk1 is shared between ph5 and ph1, fk2 between ph1 and ph2, ··· ).

```
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 2 as follows:

AE?fkid1 : cast_t[fkid1 = CreateCast("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. 3. The animations are displayed on the graphic window (Fig. 3). Fig. 3(a) shows the initial state. Fig. 3(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. 3(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. 3(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 Experimental results

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.

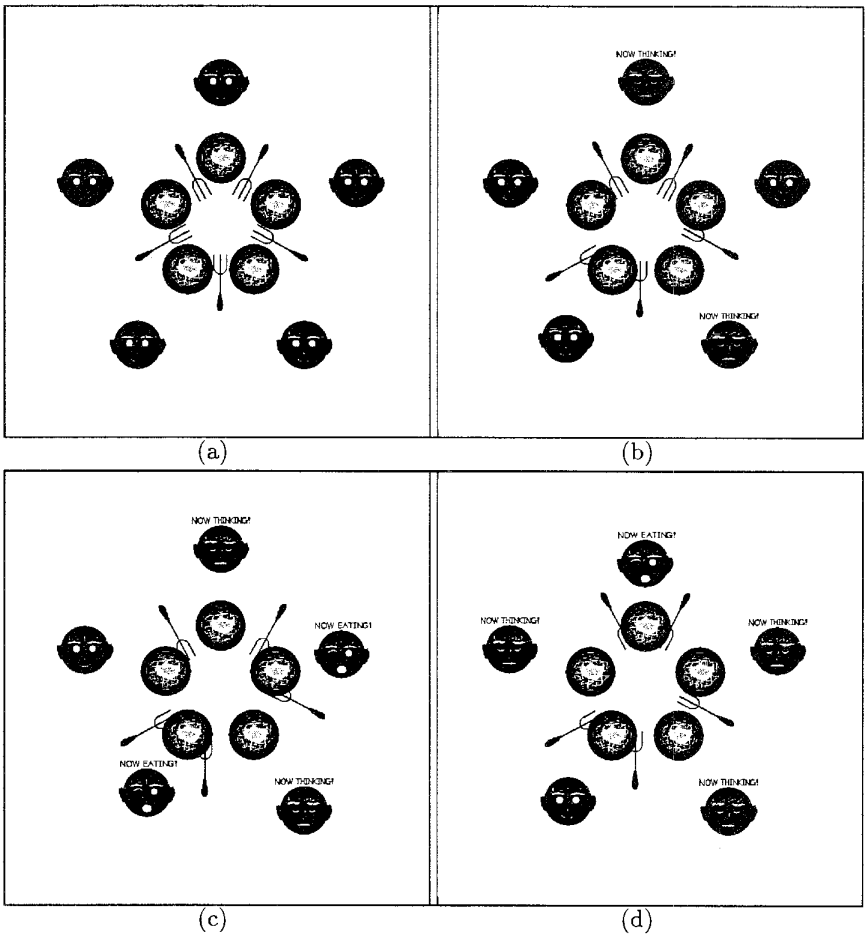


Figure 3: Visualized Dijkstra's philosophers

Table 3: Amount of description in C and LOTOS

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

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 2 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 3. 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. 4 (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.

## 6 Conclusion

In this paper, we have 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.

There are several researches for visualizing protocol specifications for facilitating understanding and designing them and for a stepwise refinement[2, 5]. In formal specification language Estelle, each process is described as a finite state machine. In Ref. [2], a visualization technique of Estelle specifications and its 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 [5] is proposed as a visual language based on LOTOS instead of G-LOTOS 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.

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.

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

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

[4] Shiroshima, T., Matsuura, T. and Taniguchi, K.: "Design and Implementation of the Interactive Animation Server", *Proc. of the 49th annual convention of IPS Japan(3R-10)* (1994) (in Japanese).

[5] 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).

[6] 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).

[7] 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).

[8] 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)*(to appear in Oct. 1995).