Protocol Animation based on Event-Driven Visualization Scenarios in Real-time LOTOS

Keiichi Yasumoto[†], Takaaki Umedu[‡], Hirozumi Yamaguchi[‡],
Akio Nakata[‡] and Teruo Higashino[‡]

[†] Graduate School of Information Science, Nara Institute of Science and Technology, JAPAN
e-mail: yasumoto@is.aist-nara.ac.jp
[‡] Graduate School of Information Science and Technology, Osaka Univ., JAPAN
e-mail: {umedu, h-yamagu, nakata, higashino}@ist.osaka-u.ac.jp

In this paper, we propose a method for animating behavior of communication protocols/systems based on event-driven visualization scenarios described in a subclass of E-LOTOS called real-time LOTOS. In the proposed method, first we pick up some interesting events in the original specification of the protocol, and then describe a visualization scenario for those events so that the corresponding animations are executed just after their execution. We execute the original specification and its visualization scenario in parallel under multi-way synchronization mechanism so that the corresponding animation is activated when each event in the original specification is executed. To describe animations in real-time LOTOS, we have defined some primitive animation operations as ADT functions. The pair of the original specification and its visualization scenario is converted into the multi-threaded Java program using our real-time LOTOS compiler where animation primitives described as ADT functions are replaced to the corresponding methods implemented as Java class libraries. In our visualization method, we can specify the visualization scenario without modifying the original specification, and we can derive a Java program which animates the original specification in real-time. We have carried out an experiment to visualize a real-time LOTOS specification of a prioritized queuing mechanism of the differentiated service where two types of visualization scenarios for algorithm animation and performance evaluation are used for visualization.

Keywords: protocol animation, LOTOS, E-LOTOS, multi-way synchronization, real-time systems, compiler

1 Introduction

Protocol visualization is useful in facilitating to understand dynamic behavior of communication protocols. It is used for the purpose of algorithm animation, performance evaluation, network management,

and so on. Several researches for protocol animation based on formal specification languages have been studied (for example, [2, 18]). A specification of a communication protocol (*original specification*) can be visualized by displaying an appropriate pre-defined animation when each event is executed. In such an event-driven 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 event occurrences and data values in I/O events. To visualize the behavior of concurrent systems in real time, it is also desirable that we can execute such visualized scenarios 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 defined a subclass of E-LOTOS[9] called *real-time LOTOS* to describe visualization scenarios of real-time protocols and developed its compiler which generates Java programs. In the proposed method, first, we describe a visualization scenario specifying animations which we would like to display when particular events in the original specification are executed. In real-time LOTOS, we can use primitive operations for animations such as registration, indication, movement and elimination of animation objects, where each of them is described as an ADT function executed with an event. Various animations can be described by combining these primitives and LOTOS operators such as choice, parallel, interruption and so on. Time constraints for protocol animation can also be specified. Then, we get the *visualized specification* by combining the original specification and its visualization scenario to be executed in parallel under the multi-way synchronization mechanism. In the visualized specification, each event in the original specification synchronizes with the same event described in the visualization scenario to activate the corresponding animation.

The multi-way synchronization mechanism of LOTOS enables dynamic data exchange among multiple concurrent processes [8]. Since data values of each I/O event can be exchanged between the original specification and visualization scenario by using the mechanism, we can select one of alternative execution paths in the visualization scenario and/or change animations depending on the timing of event execution and data values.

To evaluate the usefulness of our approach, we have developed a compiler to implement the visualized real-time LOTOS specifications as the Java programs by extending our LOTOS compiler [21] so that we can deal with both timing constraints and animations. To derive fast object code, the compiler maps multiple concurrent processes to the corresponding threads and interactions among them are implemented as the access to the shared data. To implement real-time behavior, we have implemented EDF (Earliest Deadline First) scheduling mechanism to be used in those threads. We assume that the data types and functions specified in real-time LOTOS specifications are available as the corresponding Java methods. So, our compiler provides only a mechanism to invoke those methods. Thus, our compiler can also be used as a tool for developing concurrent Java programs with a multi-way synchronization mechanism. Also we have implemented an animation server as an independent thread which updates attributes (such as location, color, and so on) of all animation objects periodically. The animation server enables more than 100 animation objects to move smoothly in parallel. Since we use Java as the implementation language, our compiler can generate Java Applets so that each visualized specification are executed on any Web browser.

In the following Sect. 2, a method for visualizing real-time LOTOS specifications is explained. Sect. 3 describes the mechanisms for executing the visualized specifications. In Sect. 4, we try to visualize real-time LOTOS specifications of a prioritized queueing mechanism of the differentiated service [3] with two visualization scenarios for algorithm animation and performance evaluation. Some experimental results are shown in Sect. 5. In the final Sect. 6, we conclude the paper.

1.1 Related work

There are several approaches for visualization of systems, and they are roughly classified into three categories: (1) algorithm animation methodology, (2) visualization languages/tools, and (3) protocol animation.

For category (1), there are several researches [4, 11]. [11] discusses about essential characteristics in visualizing concurrent systems and proposes a paradigm which is used in their visualization tool set called Parade [12]. [4] has proposed a technique based on shape graphs where the dynamic change of internal data structures such as heap area is animated for understanding of the given algorithm.

For category (2), many visualization languages and tools have been proposed [1, 5, 12, 17]. Parade [12] consists of a scheduler for displaying animation objects efficiently, a translator to map program events and parameters onto animation actions, and a controller that supports to visualize multiple execution sequences at the same time. In VERDI system [5], a graphical language which has a facility of multi-party interactions is used. The system behavior must be written by using a graphical editor. DisCo [1] is a formal description method for reactive and distributed systems and later extended to support real-time features. It uses an object-based programming language for specifications, and contains a complier in DisCo tool-set can translate given specifications into Java packages called *specification engines.* Using the animator in the tool-set that offers visualization windows as well as an interface between the specification engines and designers, the execution of the specifications is animated where message exchanges between objects are presented as MSCs.

In all of the above languages and tools, we must describe the behavior of the system which we would like to visualize in their own graphical language. On the other hand, in the proposed method, we can visualize the existing protocols without modifying the original specifications of those protocols, although they must be written in real-time LOTOS. It is the main difference between the above visualization methods and the proposed one.

For category (3), there are several FDT based researches [2, 18]. In formal specification language Estelle, each process is described as a finite state machine. In [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 in a graphical 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 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 [18] 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 difficult.

2 How to animate protocol behavior

In order to describe protocols and the corresponding animations, we use a subset of E-LOTOS[9], which we call *real-time LOTOS*.

Table 1: Example of multi-way synchronization specified among three processes

```
  P1[a,b] |[a,b]| (P2[a,b] |[a]| P3[a,b])
where
  process P1[a,b]:noexit:=
     a!1; ...
  [] a!0; ...
  [] b!0; ...
  endproc
  process P2[a,b]:noexit:=
     b?y[g(y)]; a!f(y); ...
  [] a?x; ...
  endproc
  process P3[a,b]:noexit:=
     a?z[h(z)]; ...; b?w; ...
  endproc
```

($f(y), g(y)$ and $h(z)$ are ADT functions.)

2.1 Real-time LOTOS

In standard LOTOS language [8], a system is modeled by a set of several concurrent processes where the behavior of each process is defined by sequences of *events*. Each event occurs at an interaction point called *gate* with input/output of values. `a?x:int` denotes an event which inputs a value from gate `a` and substitutes the value to variable `x:int`, while `b!Exp` denotes an output action to gate `b` where the value of `Exp` is output. In order to specify temporal ordering of events, the action-prefix (`a;B`), the choice operator (`B [] B`), the parallel operator with/without synchronization gates (`B |[gl]| B`, `B ||| B`), the disabling operator (`B [> B`), and the enabling operator (`B >> B`) can be specified between any two behavior expressions `B`. Also, with `[boolexp]-> B`, we can specify to execute `B` if the boolean expression `boolexp` holds. With the parallel operator (`|[gl]|`) with synchronization gates(`gl`), we can specify multiple concurrent processes to execute some events and exchange data values in synchronization with each other (called *multi-way synchronization*).

In $P1|[a,b]|(P2|[a]|P3)$ of Table 1, events with gate a have to be executed synchronously among P_1, P_2 and P_3. For example, when P_1, P_2 and P_3 execute $a!0$, $a?x$ and $a?z[h(z)]$, respectively, the output value "0" of $a!0$ must be assigned to the input variables x and z (the variable types of x and z must match the type of "0" and $h(0)$ must hold). When P_1 and P_3 execute $a!1$ and $a?x$, respectively, and P_2 executes $a!f(y)$, the value $f(y)$ must be equal to the output value "1". When multiple synchronizations become executable simultaneously, one of them must be selected and executed exclusively. For example, in this example, when P_1 executes $b!0$, either P_2 or P_3 can execute the event on gate b ($b?y$ or $b?w$).

In real-time LOTOS, some new constructs of E-LOTOS can be used in addition to operators of the standard LOTOS such as choice, parallel, multi-way synchronization and disabling. In Table 2, we show the new constructs of E-LOTOS which we have adopted in real-time LOTOS, where we can specify the time at which each event may be executed, explicit iterations of some behavior [7], write-many variables, and variable assignments. Here, $a@?t$ captures the time duration (called *elapsed time*) until the event is

Table 2: New constructs introduced to real-time LOTOS

syntax	semantics
loop B **endloop**	iterate B.
while E **do** B **endwhile**	iterate B while E holds.
var $V(,V)^*$ **in** B **endvar**	declare write-many variables used in B.
$a@?t$	substitute the elapsed time (since the last event was executed) to variable t.
$a@?t[p(t)]$	a may be executed at elapsed time t where $p(t)$ holds.
$a@!T$	a may be executed at elapsed time T.
wait(d);B	delay execution of B by d time.
$?v := E$	variable assignment

executed since it was offered (i.e., since the last event was executed in the same process). The timing constraint of an event is given by a guard expression which specifies time ranges in which the event may be executed. Event $a@?t[constraint(t)]$ means that this event can be executed at elapsed time t such that $constraint(t)$ holds. We omit detailed explanation about the timed semantics in real-time LOTOS, since it is similar to that of E-LOTOS and ET-LOTOS[13, 14].

Restrictions about time

To simplify the implementation of real-time behavior which we will explain in Sect. 3, we assume the followings in this paper.

- the type of time is represented by integer numbers. We assume that each time unit corresponds to exactly 1 msec. However, if we would like to execute given specifications more slowly, we can assign any time interval to a unit of time when compiling the specifications.
- each timing constraint must be a time range like $C_1 \leq t \leq C_2$. Here, each time range can be given as any expression whose form is a logical product of multiple inequalities. Each time range need not be closed (e.g., $C_1 < t$ can be specified). However, if a given timing constraint can be represented statically as a finite set of time ranges, we can describe it as an equivalent choice statement. For example, $a@?t : time[C_1 \leq t \leq C_2 \ or \ C_3 \leq t \leq C_4]$ can be described as $a@?t_1 : time[C_1 \leq t_1 \leq C_2]$ [] $a@?t_2 : time[C_3 \leq t_2 \leq C_4]$.

2.2 How to describe scenarios

In this paper, we use multi-way synchronization to activate the corresponding animation when an event is executed. The basic idea is as follows:

For a given original specification of a protocol in real-time LOTOS,

- specify the interesting events and their corresponding animations in a visualization scenario,
- execute the original specification and its visualization scenario in parallel using multi-way synchronization.

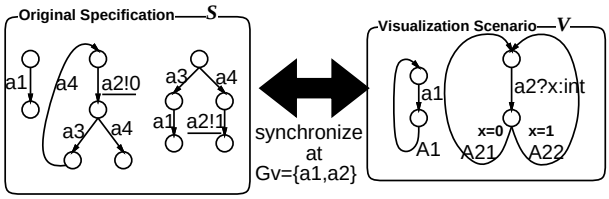


Figure 1: Our visualization method

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 where we would like to visualize events on those gates.

Let $S[E]$ be the behavior expression of an original real-time 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 $G_V = \{a_1, ..., a_n\} \subseteq E$, we specify the visualization scenario $V[G_V]$ for $S[E]$ as follows (here, $\prod_{k=1}^{n} V_k$ denotes $V_1|||...|||V_n$):

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

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

Here, A_k is an animation for a_k, which is a process composed of animation events explained in Sect. 2.3. If we would like to specify the events in $G' \subseteq G_V$ so that each event in G' and its corresponding animation must be finished 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 $V_1[]...[]V_n$):

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

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

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

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

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[8]. 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[8] 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.

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

Using the mechanism, the data values of each input/output in the specification can be transmitted to the visualization scenario, and different animations (one of A_{11}, A_{21}, A_{22} and A_{23}) can be displayed depending on the values. The above technique can be also used if several values are input/output in an event. Moreover, if we would like to change the progress of visualization depending on the execution time of an event, the similar technique can be used.

2.3 Describing animations

In order to describe animations in real-time LOTOS, first we introduce some primitives for animation operations such as registration, indication, movement and elimination of animation objects. Hereafter, we call each animation object as *cast*.

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

$$?id := Operation(parameters)$$
or
$$dummy!Operation(parameters)$$

Here, id is a variable for keeping an identifier of each cast/window whose sort is *cast_t/window_t* (variable id must be declared by var construct). We use *dummy* as a dummy gate for animation operations which have no return values. Here, gate *dummy* is defined by **hide** operator.

For example, we describe an operation which registers a JPEG file "packet.jpg" as a cast used in an animation as the following variable assignment ($wid, posx$ and $posy$ are a window ID, and the initial position in the window, respectively).

Table 3: Animation primitives

Primitives	Contents
CreateWindow	creating a window for displaying animations
CreateCast	registering an object as 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
...	...

Table 4: Sorts of casts

Sorts	Options
image	filename
string	string, font size
circle	radius,color
rectangle	width,height,color
...	...

$$?\mathrm{cid} := \mathrm{CreateCast}(\mathrm{wid}, \text{"image : packet.jpg"}, \mathrm{posx}, \mathrm{posy})$$

We show the part of animation primitives in Table 3. The sorts of the casts are specified by strings as "sort:option,...". We define some sorts in Table 4.

Here, each animation event using, say, "MoveCast" primitive takes non-zero time to finish animation behavior, and it may not seem to match LOTOS semantics. For this matter, we assume that each animation event finish its execution immediately at time when it starts. Therefore, the behavior of an animation event which takes 4 units of time can be considered as $i; wait(4)$.

Using operators of LOTOS such as choice, disabling, and parallel, we can describe various animations such that the multiple casts are moving in parallel. Also, by specifying time constraints to animation events, we can easily control the duration of each animation and/or the whole animation behavior.

3 Executing visualized specifications

We use Java language to implement the visualized specifications. For the purpose, we have implemented a compiler which generates the corresponding Java programs from given real-time LOTOS specifications.

3.1 Compiling Real-time LOTOS into Java programs

In this paper, based on a technique proposed in [20], we implement given real-time LOTOS specifications as the corresponding Java programs in the following steps.

1. transform a given real-time LOTOS specification to an intermediate program in *concurrent synchronous EFSMs model (S-EFSMs)* [10, 20] where S-EFSMs consists of multiple concurrent EF-

	E_1	E_2	E_3
p_1	$(a!0,$	$a?x$	$a?z[h(z)])$
p_2	$(a!0,$	$a!f(y)$	$a?z[h(z)])$
p_3	$(a!1,$	$a?x$	$a?z[h(z)])$
p_4	$(a!1,$	$a!f(y)$	$a?z[h(z)])$
p_5	$(b!0$	$b?y[g(y)])$	
p_6	$(b!0$		$b?w)$

Table 5: Possible instances

	P_1	P_2	P_3
r_1	$(\{a!0\},$	$\{a?x,a!f(y)\},$	$\{a?z[h(z)]\})$
r_2	$(\{a!1\},$	$\{a?x,a!f(y)\},$	$\{a?z[h(z)]\})$
r_3	$(\{b!0\},$	$\{b?y[g(y)]\})$	
r_4	$(\{b!0\},$		$\{b?w\})$

Table 6: Multi-rendezvous table

SMs and a multi-rendezvous table which is a compressed information containing combinations of synchronizing EFSMs, a tuple of synchronous events and guard expressions.

2. transform each EFSM to the corresponding Java code and invoke it as a single Java thread. Multi-way synchronization among threads is implemented using a shared table (variables) representing the multi-rendezvous table where each thread writes information about synchronous events which the thread wants to execute.

3. schedule execution order of events among threads depending on the deadlines extracted from time constraints attached to the events. EDF (Earliest Deadline First) policy is used.

3.1.1 Transformation to Concurrent Synchronous EFSMs

Concurrent Synchronous EFSMs

In S-EFSMs model, we suppose that each EFSM has a finite number of registers (variables), that an execution condition called the *guard* expression can be specified to each transition, and that each transition can input/output several values via a gate as an I/O action. Each transition is denoted by $g\ v\ [f]$ where g is a gate, v is a sequence consisting of input variables ($?x\ :\ t$) and output expressions ($!E$) on gate g, and f is a guard expression of the transition.

In S-EFSMs model, we statically derive the following information about synchronizations to simplify the dynamic evaluation : (1) all possible tuples of synchronizing events and the corresponding tuples of EFSM names, (2) the execution condition (guard) for each tuple. For the example of Table 1, we show all possible instances of synchronization tuples in Table 5. Here, $p1, ..., p4$ show the possible instances on gate a, and $p5$ and $p6$ show those on gate b.

Since the number of all possible instances of synchronization tuples may become large in general, we adopt a compressed notation where multiple instances are represented as one tuple called *rendezvous indication*. Each rendezvous indication consists of (1) a tuple of EFSMs/event sets where each set includes events executed in an EFSM and (2) the execution condition. Here note that we compose each tuple so that any combinations of events constructed by selecting one event from each event set can satisfy the synchronization condition and guard, although events with different output values are assigned to separate rendezvous indications. The set of all rendezvous indications is called *multi-rendezvous table*. In Table 6, we show the multi-rendezvous table for the example of Table 1.

Transformation algorithm

First we decompose a given real-time LOTOS specification to the parallel composition of multiple *sequential behavior expressions (SBEs)* where each SBE contains only the action prefix ($a; B$), choice ($B1[]B2$) and the jump operations to any location within the SBE (denoted by **label** L:B and **goto** L). Outline of the transformation algorithm is as follows (for the details, see [20]).

- For the parallel composition (B1$|[G]|B2, B1|||B2$), decompose it to sub-behavior expressions: $B1$ and $B2$.
- For the action prefix like $a;(P|[G]|Q)$ where several parallel processes are invoked after sequential behavior, transform it to the equivalent parallel composition like $hide\ \gamma\ in\ a;\gamma;P|[G \cup \gamma]|\gamma;Q$, and decompose it to sub-behavior expressions.
- For the process invocation, replace it with the behavior expression of the process after inserting **label** *process_name:*. If the behavior expression of the same process has already appeared somewhere in the current behavior expression, just insert **goto** *process_name*. **loop** and **while** statements are similarly replaced by **label** and **goto**.
- For the enabling sequence ($B1 >> B2$) where each sub-behavior expression contains parallel operators like $B1 := B11|||B12$ and $B2 := B21|||B22$, transform it to the equivalent parallel composition like $B1'|[\alpha]|B2'$ where $B1' := B11 >> \alpha; B21$ and $B2' := B12 >> \alpha; B22$ and decompose it to $B1'$ and $B2'$. For the choice ($B1[]B2$) and the disabling ($B1[> B2$), the similar technique can be applied to decompose each of them to sub-behavior expressions.
- For the disabling ($B1[> B2$) where $B1$ and $B2$ are both SBEs, transform it to a single SBE by representing the disabling operator by several choice operators. For example, $a;b;exit[> c;exit$ can be transformed to $a;(b;(exit[]c;exit)[]c;exit)[]c;exit$.

In order to make it possible to apply the above algorithm, we impose the following restrictions on process instantiations in given real-time LOTOS specifications.

- Recursive processes are allowed when they are tail recursion (e.g. $P := B >> P$).
- Recursive processes which may produce infinite behavior such as $P := (B1 >> P >> B2)[]exit$ or $P := B\ op\ P$ ($op \in \{[], |[G]|, [>\}$), are not allowed. However, if the recursive process call is guarded and the guard expression can be evaluated statically (e.g. $P(x) := B|||([x < 100]- > P(x+1))$), we treat such a process.
- Mutually recursive processes are allowed as long as process calls are guarded and the guard expressions can be evaluated statically.

By applying the above transformation algorithm to a given real-time LOTOS specification, we can get the parallel composition of multiple SBEs. Each SBE can easily be transformed to an EFSM due to its nature. From the syntax tree of the parallel operators specified among SBEs, a multi-rendezvous table can automatically be calculated. Here, due to space restriction, we omit the details (the details can be found in [20]).

3.1.2 Implementation of concurrent synchronous EFSMs

It is quite easy to implement each EFSM as a single Java thread if it contains no time constraints nor multi-way synchronization. In that case, each thread just keeps its current state, selects one of events executable at the current state, and changes the current state after executing the selected event. For the selection, an event is selected based on random numbers among candidate events whose guard expressions and other environmental conditions (e.g., input values are available at the specified gate) hold.

We have implemented the multi-way synchronization mechanism among threads using a multi-rendezvous table as follows. Below, we call the data structure for a multi-rendezvous table as *SyncTable*.

(1) For each event the thread can execute at its current state, it writes READY flag to the corresponding location of the rendezvous indication in SyncTable with its guard expression and output values (e.g., the value of E in event $g!E$). If there are many rendezvous indications containing the event, it writes the same information to all the rendezvous indications.

(2) After step (1), if there are a rendezvous indication where all participant threads have written READY flags and all guard expressions hold, it concludes that the event can be executed by the rendezvous indication. If there are several such rendezvous indications, it selects one at random. If there is no such rendezvous indication, it sleeps until such indication appears (it is waked up by the signal sent from the other thread).

(3) When the thread finds at least one executable rendezvous indication in step (2), it executes the event and removes all information written in step (1) from SyncTable. It also sends the signal to all the other threads to inform that the rendezvous indication is executable.

Since all of the above steps must be executed exclusively among threads, we have used `synchronize` property of Java when calling the Java method corresponding to the above algorithm.

3.1.3 How to implement real-time behavior

Since Java thread mechanism has no facility to handle real-time behavior, we simulate EDF (Earliest Deadline First) scheduling using a shared table called TimeTable by the following algorithm.

(1) Each thread calculates the starting time and deadline of events which are executable at the current state, registers them to TimeTable, and sorts the contents of TimeTable in increasing order of deadlines.

(2) It sleeps until its entry comes to the top of TimeTable.

(3) After executing the event, it removes the starting time and deadline from TimeTable, and repeats the above procedure from step(1) for the next event.

Below, we explain how to implement time constraints attached to event sequences, their choices and multi-way synchronization, using the above EDF scheduling.

Assignment of time parameters to real-time threads

When we use EDF scheduling, if a thread T wants to start the execution of a task B at time t_1 and to finish it before time t_2, the following procedure should be done by T itself: (1) set the starting time and deadline to t_1 and t_2, respectively, before yielding the CPU to the other threads, (2) execute B when T is scheduled, and (3) check whether B has been executed in time. In our real-time LOTOS, since each timing constraint attached to an event is represented as a range $C_1 \leq t \leq C_2$ by the restriction in Sect. 2.1, the constraints can be easily mapped to the starting time and deadlines of threads. In the following, we explain how to implement each EFSM with timing constraints in a real-time thread [1].

Action prefix sequence

[1] In the explanation, relative time is used for starting time and deadline of each thread although absolute time is used in the thread mechanism

A thread to which an event sequence of $a_1@?t_1[s_1 \leq t_1 \leq e_1]; \cdots; a_n@?t_n[s_n \leq t_n \leq e_n]$ is assigned, sets its starting time and deadline to s_1 and e_1, respectively. After the thread is scheduled around time s_1, it executes the event by receiving some values (if the event is like $a?x$), by finding the synchronization peers (if the event is specified to synchronize with others) or by finishing calculation (if the event is like $a!f(x)$) before time e_1. The same procedure is applied to the next event a_2.

If $wait(d)$ operator is inserted between events like $a; wait(d); b@?t[C_1 \leq t \leq C_2]$, the delay d is added to the starting time and deadline of the event just after $wait(d)$ like $a; b@?t[C_1 + d \leq t \leq C_2 + d]$.

Choice among event sequences

If several alternative timed events become enabled, we give a higher priority to the event whose timing constraint expires earlier.

Here we suppose that an expression $a_1@?t_1[s_1 \leq t_1 \leq e_1]; B_1 [] \cdots []a_n@?t_n[s_n \leq t_n \leq e_n]; B_n$ is assigned to a thread. Let a_{min} be the event with the earliest deadline (denoted by e_{min}) in all event candidates $a_1, \cdots, a_{n-1}$ and a_n. Let s_{min} be the earliest starting time in all events.

Our implementation is as follows: the thread first sets its deadline to e_{min}. Similarly to action prefix sequences, the thread sets its starting time to s_{min}.

After the thread is scheduled around time s_{min}, it waits an event to become executable by receiving some values, finding the synchronization peers or finishing calculation. If a_{min} has not been executed until e_{min}, then a_{min} is excluded from the candidates, and the thread delays its deadline to the second earliest deadline in the other events. If the event a_i is executed within its timing constraint, the above procedure is applied to the succeeding expression B_i.

3.1.4 Scheduling

To execute timed events before their deadlines, the threads to which EFSMs are assigned are scheduled by EDF policy. For interactions among threads, we also need the mechanisms for completing execution of each event before its timing constraint expires, for scheduling threads to reach the synchronization points in time, and for controlling the priority among multiple combinations of synchronizing event tuples.

Synchronization among timed events

Hereafter, we call each tuple of events to be executed synchronously as a *rendezvous*.

To implement multi-way synchronization in real-time LOTOS, we need to check the synchronization condition [2]. In addition, the generated object code should have the following mechanisms: (i) priority control among multiple timed rendezvous according to their starting time; and (ii) timing control for synchronizing threads so that they can reach the synchronization points in time.

(i) Priority control among multiple timed rendezvous

$P|[a]|Q$
$where$
$P := \cdots; a@?t[1 < t < 3]; \cdots,$
$Q := \cdots; a@?t[2 < t < 5]; \cdots$

In the above behavior expression, let us assume that event a becomes enabled at time t_p in P and at time t_q in Q, respectively. Then, a can be executed at time t such that $Max(t_p + 1, t_q + 2) \leq t \leq$

[2] Multi-way synchronization among a tuple of events can be executed if and only if, for each pair in the tuple, the number and types of their I/O parameters are identical, their guard expressions hold for values exchanged between those events [8] and there is a common time range between their timing constraints.

$Min(t_p + 3, t_q + 5)$. In this case, if the threads for P and Q have calculated the rendezvous before time $Min(t_p + 1, t_q + 2)$, they need to wait until then before executing the rendezvous. However, another alternative rendezvous with earlier executable time may be executed while they are waiting. Therefore, we have introduced a queue which keeps the calculated rendezvous in increasing order of their starting time. When each thread has calculated an executable rendezvous, it registers the rendezvous in the queue and sorts its entries. Then, the thread sets its starting time to the starting time of its rendezvous and yields the CPU to the other threads. When the thread is scheduled at its starting time, it checks whether its rendezvous is the first entry in the queue or not. If so, it executes the rendezvous and excludes the other alternative rendezvous in the queue so that they cannot be executed after that.

(ii) timing control for synchronizing threads

In our implementation, we use EDF scheduling and a rendezvous becomes executable only when all of its participant threads have written READY flags to the corresponding rendezvous indication of the multi-rendezvous table. Therefore, if a rendezvous consists of some real-time threads with deadlines and other non real-time threads without deadlines, the latter threads may not be scheduled (due to low priority) and the rendezvous may not be executable forever.

Here, we have adopted a simple solution, which is to give the highest priority (e.g., 0 for deadline) to the non-real-time thread temporally when some synchronizing events are enabled in the thread. With this facility, the synchronization condition can be checked among any kinds of threads prior to real-time tasks in other threads.

For the rendezvous between a non-timed process consisting of iteration of non-timed events and a periodic process which executes some events in a certain time period, we can statically decide the tuples of synchronizing events between them. In this case, for each synchronizing event, the thread for the non-timed process sets its deadline to the deadline of the corresponding event in the periodic process so that the former thread can reach the synchronization point before that time by completing the preceding behavior block.

$P \ |[b_1, \cdots, b_k]| \ Q$
where
$P :=$ loop $B_1 >> b_1; \cdots; B_k >> b_k$ endloop
$Q :=$ loop
$b_1@?t_1[t_1 \leq T_1]; \cdots; b_k@?t_k[t_k \leq T_k];$
endloop
(here, each B_i is an action prefix sequence)

For example, in the above specification, execution of each B_i in P should finish before the corresponding b_i in Q becomes enabled. So, the thread for P sets its starting time and deadline to 0 and T_i when $B_i >> b_i$ becomes enabled. This facility is essential on efficiently executing timed multi-way synchronization among multiple processes where timing constraints are specified in the constraint oriented style[19].

Example of timed multi-way synchronization execution

Fig. 2 shows how one of two alternative timed rendezvous is selected and executed. Let T_s be the time when invoking the initial behavior expression. The two threads for processes P and Q are invoked at time $T_s + 1$, and the thread for R at time $T_s + 2$. Let us suppose that the threads corresponding to P, Q and R are scheduled in this order. We refer those threads simply by P, Q and R.

First, thread P stores READY flag to the corresponding rendezvous indications, say, $r1$ and $r2$ in SyncTable and sleeps until receiving a signal from other threads. After that, thread Q stores READY flag

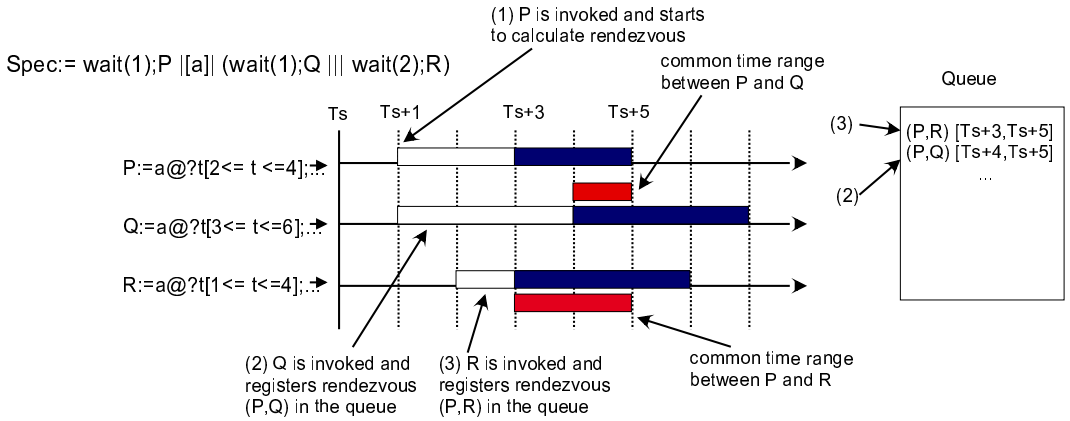


Figure 2: Example of timed multi-way synchronization execution

to $r1$ in SyncTable and detects that $r1$ can be executed between P and Q in the time range $[T_s+4, T_s+5]$. Then it stores $r1$ with the time range in the queue, and sleeps until reaching $T_s + 4$ point of time.

After thread R is invoked at T_s+2, it stores READY flag to the rendezvous indication $r2$ and detects that $r2$ is executable between P and R in the range $[T_s + 3, T_s + 5]$. So, it registers $r2$ with the range in the queue and sleeps until reaching $T_s + 3$ point of time. Rendezvous indications $r1$ and $r2$ in the queue are sorted in increasing order of their starting time.

When thread R is waked up at time $T_s + 3$, it executes its event of $r2$ and sends a signal to wake P up since it is the first entry of the queue. At the same time, rendezvous indication $r1$ between P and Q is removed from the queue because it is mutually exclusive to the first one. When thread Q is scheduled at $T_s + 4$, it knows it cannot execute its event of $r1$ by referring the queue.

3.1.5 Animation primitives

We have also extended our real-time LOTOS compiler to treat the animations. We have implemented an interactive animation server explained in the next section to display the animations in the generated object codes. Since most of the animation primitives defined in Table 3 are equivalent to the instructions on the animation server, our compiler only replaces the animation primitives used in events to invocations of the corresponding Java methods.

3.2 Mechanism for real-time animation

Our Animation Library enables an easy operation for each animation object(*cast*) such as registration, indication, modification of attributes (*i.e.* location, scale, color, and so on) and elimination of it. To show animations continuously according to a scenario using Animation Library, it is needed to repeat the following process:

- The user process (i.e., each real-time LOTOS process) sends commands for changing cast attributes to Animation Server in advance. Even when multiple processes send such commands simultaneously, the Animation Server can accept those commands by serializing them.
- The Animation Server periodically updates the animation window (*Panel*) so that all modifications for cast attributes during the current time period are reflected.

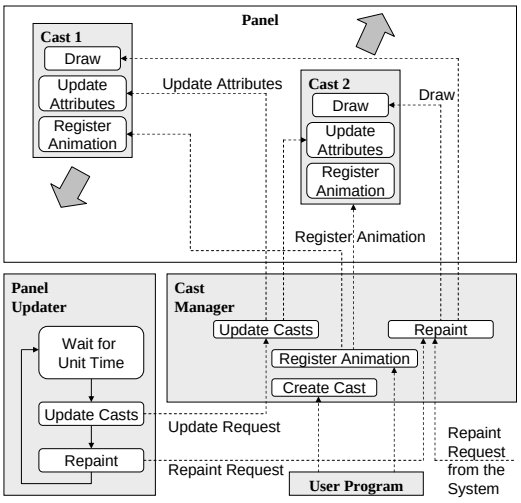


Figure 3: Animation server

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 Panel for casts should be updated at a fixed time interval.

We have composed the mechanism of three types of modules: (1) a Panel Updater module for updating and repainting the Panels, (2) a Cast Manager module for managing all the animation and (3) a Cast module for managing each cast. For simplicity, we fix the time interval for updating the Panel to a certain constant such as 30 frames/sec (here, we call each image displayed on the Panel at every time interval as *frame*).

When the multiple modules for moving casts are executed in parallel, they and the Cast Manager module are synchronizing at a fixed time interval as the following steps (Fig. 3).
[Behavior of a Panel Updater module]

1. calling a method of the Cast Manager module to set the attributes (location, scale, appearance, and so on) of all the casts as registered until the time.
2. updating the Panel by calling a method of the Cast Manager module to repaint the Panel.
3. waiting until the next time interval and repeating the above process.

[Behavior of a Cast Manager module]

1. setting the time to all the cast when called from Panel Updater module.
2. repainting all the casts to update the Panel when called from Panel Updater or the System.
3. creating new casts or registering the animation of the casts when called from user processes.

[Behavior of a Cast module]

1. managing the registered animation of the cast.

2. calculating the location and the other attributes at the time when the time is set by the Cast Manager.

3. drawing the cast when requested by the Cast Manager module.

Since a Panel Updater module is implemented as a thread in our library, each animation can be executed in real time. Using the library, the dynamic behavior of the multiple concurrent processes can be visualized.

3.3 Analysis of protocol behavior based on real-time traces of events

Our compiler can generate the object program which outputs real-time traces of events in the following format.

$$TR := a_1!E_1!..@!t_1; a_2!E_2!..@!t_2; ...; a_n!E_n!..@!t_n$$

Here, each trace is a sequence of events connected by ";" where each event is described with the gate name, the output value(s) and the time at which the event was executed.

In Sect. 2, we explained that we compose a visualized specification as $S \ |[G_V]| \ V$ where S and V are the original specification and the visualization scenario, respectively. Suppose that we just execute the program of S and get its real-time trace $TR(S)$. By executing the program for $TR(S)|[G_V]|V$, we can easily play back $TR(S)$ with visual animations. This will help designers/programmers to find and correct bugs in parallel programs. Also, by changing the scale for a unit of time when compiling or just by editing timing constraints in $TR(S)$, we can speed up/down the animations flexibly.

4 Example of visualization

In recent years, the Differentiated Service (DiffServ)[3] technology has been focused on as a QoS control mechanism for flows on the Internet. In DiffServ, packets are marked and classified to several classes in the Ingress router so that some kind of prioritized processing is applied to the packets depending on their classes as shown in Fig. 4(e.g., differentiating packets for real-time traffic like VoIP from normal traffic like FTP). The Core router provides prioritized queuing for packets such as DRR (Deficit Round Robin)[16] and WFQ (Weighted Fair Queuing)[15] instead of classic FIFO queuing.

In this section, we describe a specification and the corresponding visualization scenario of such a prioritized queuing mechanism (DRR is used here) using real-time LOTOS.

4.1 Description of a prioritized queuing mechanism

We compose the DRR mechanism of three parts: (1) $PacketIn$, (2) $Queue(i)$, and (3) $PacketOut$. Here, $MaxClass \ Queue(i)$ modules are executed in parallel where $MaxClass$ is the maximum number of classes, and $1 \leq i \leq MaxClass$. net_in and net_out are gates which correspond to an input link from the network to the router and an output link from the router to the network, respectively. a and b are internal gates which are used as interaction points between $PacketIn$ and $Queue(i)$ and between $Queue(i)$ and $PacketOut$, respectively.

In Fig. 5, we show the example specification of DRR in real-time LOTOS (here, state diagrams are used due to space restriction). Communication among these processes are specified with parallel operators of real-time LOTOS as follows:

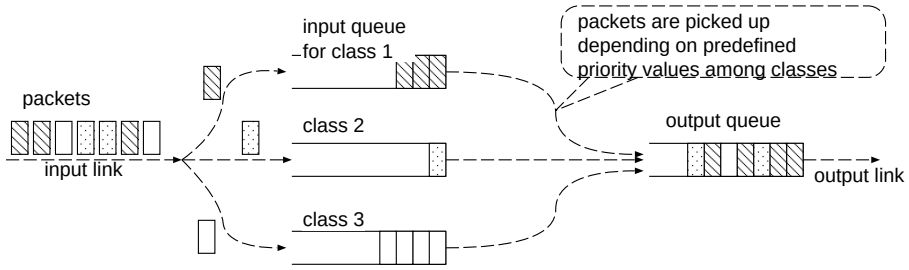


Figure 4: Prioritized Queuing

$$PacketIn[net\_in, a]||[a]|(Queue[a,b](1)|||...|||Queue[a,b](MaxClass))||[b]|PacketOut[b, net\_out]$$

As shown in Fig.5, first process $PacketIn$ inputs a packet ($net\_in?pct$) and sends it to process $Queue(i)$ with its class number ($a!class(pct)!pct!..$). Process $Queue(i)$ receives the packet via gate a and stores it to its queue ($queue$) only if the packet's class is i and $queue$ has a space to accommodate the packet ($a!i?pct!OK[size(queue) + size(packet) \leq Qsize]$). If there is no space, the packet is dropped ($a!i?pct!Drop[...]$). If $queue$ is empty, process $Queue(i)$ offers event $b!i!Empty$. If $queue$ is not empty, $Queue(i)$ sends the packet on top of the queue ($Head(queue)$) to $PacketOut$. Process $PacketOut$ receives the packets from $Queue(1), ..., Queue(MaxClass)$ and transmits to the output link (net_out) in round robin manner. When the size of the packet from $Queue(j)$ is equal or less than the deficit counter ($b!j?pct!OK[size(pct) \leq Dcnt(j)]$), it is transmitted and the counter value is decremented by the packet size ($?Dcnt(j) := Dcnt(j) - size(packet)$). The packets with class j are transmitted by repeating this until the counter value becomes less than the packet size. When the deficit counter is less than the packet size, the packet is not transmitted in this round ($b!j!pct!WT[..]$), and the counter value is increased by $F * Rate(j)$ for the next round where $Rate(j)$ is the ratio for class j's packets to the output link capacity, and F is the average number of bytes which are output to the link in one round. Then the above process is repeated for class $j + 1$.

In the next sections, we visualize specification DRR using two scenarios for algorithm animation and performance evaluation.

4.2 Visualization scenario for algorithm animation

In order to visualize specification DRR, we must describe a visualization scenario where the corresponding animation is activated when a particular event in DRR is executed.

Fig. 6 shows an example visualization of DRR where four subsequent snapshots showing (i) the situation that the queue has been full by too many packet arrivals, (ii) the situation when a packet of class 2 has arrived, (iii) the situation that the arriving packet is dropped, and (iv) the situation that the congested state has gone. Here, we will show how to describe such a visualization scenario of specification DRR.

Assume that given specification DRR is executed in the following environment with processes $Network1$ and $Network2$ where $Network1$ and $Network2$ simulate how packets arrive and are output, respectively (i.e., $Network1$ creates packets and sends it to gate net_in and $Newtork2$ receives packets from gate net_out and consumes them).

```
Network1 |[net_in]| DRR |[net_out]| Network2
```

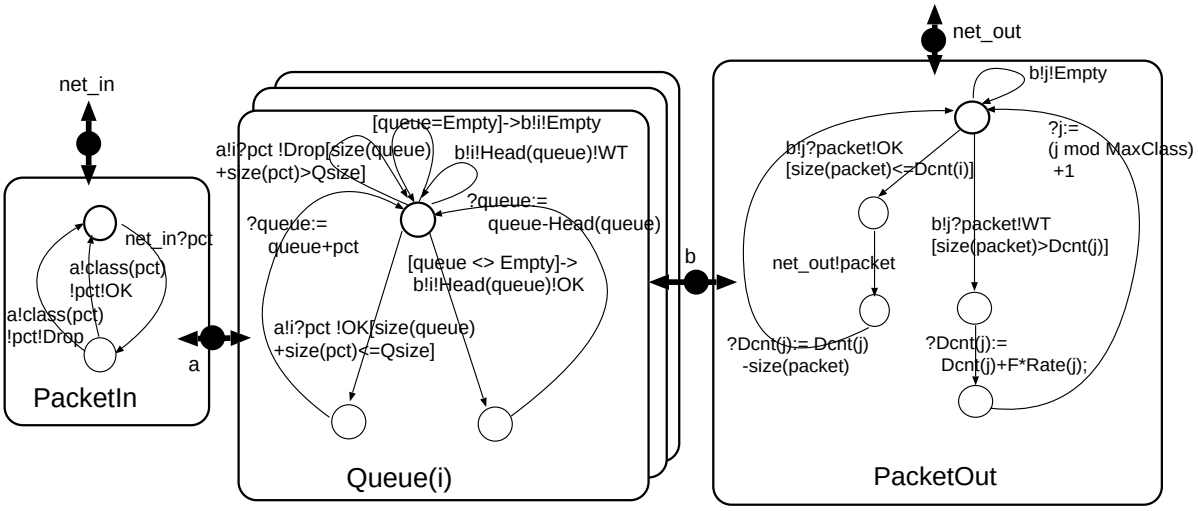
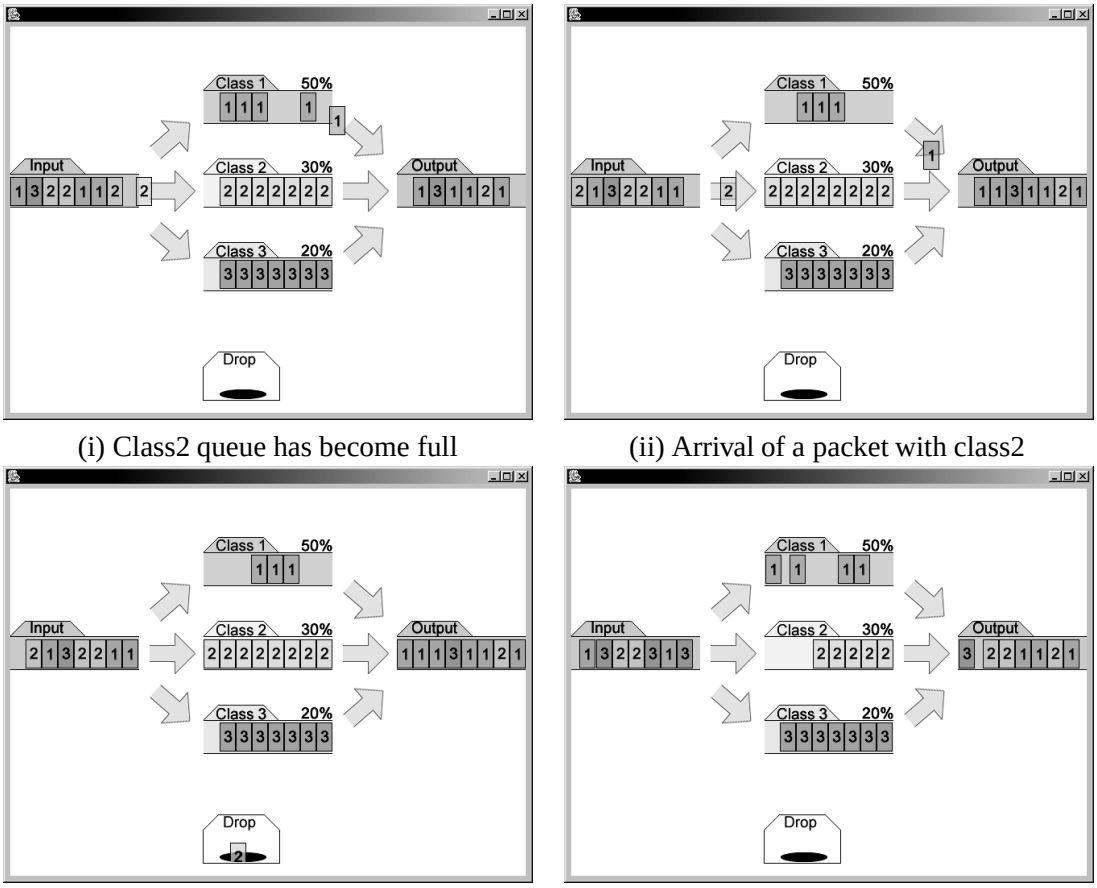


Figure 5: Specification of a prioritized queuing algorithm (DRR)



(i) Class2 queue has become full

(ii) Arrival of a packet with class2

(iii) Dropping of the packet

(iv) After congested state has gone

Figure 6: Visualization of a prioritized queuing mechanism

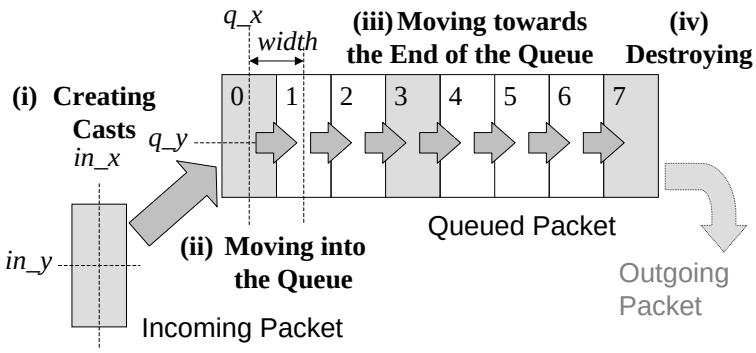


Figure 7: Animation of each queue

In this case, we can add new visualization scenario VS to this DRR specification as follows.

```
Network1 |[net_in]| (DRR || VS) |[net_out]| Network2
```

This new specification $DRR||VS$ should be able to behave similarly to the original specification DRR except for displaying animations.

First, we must extract some specific events to be visualized from the original specification and consider how to visualize them to describe animations corresponding to the events. When there are n such events, the scenario will look like the following.

```
VS:=
  loop
       evnet_1; (animation of event 1)
  []   evnet_2; (animation of event 2)
  []   event_3; (animation of event 3)
  []   ...
  []   event_n; (animation of event n)
  endloop
```

Suppose that we would like to observe the following events in specification DRR.

1. arrival of a packet from the incoming link of the network ($net\_in?packet$)
2. storing a packet into each class queue ($a?class?pct!OK$)
3. dropping of a packet due to no more room in a queue ($a?class?pct!Drop$)
4. extracting a packet from a queue and transmitting it to the outgoing link ($b?class?pct!OK; net\_out!pct$)

Next, we design a visualization scenario for each extracted event. For example, we explain the case for animating (1) arrival of a packet, (2) storing a packet into each class queue, and (4) extracting and transmitting a packet. Fig.7 shows the overview of an example animation. Here, we assume that the maximum number of packets in each queue is at most eight. The animation can be composed of four animation operations: (i) creating a cast representing the packet and displaying it at the initial position,

(ii) moving the cast into the entrance position of the queue, (iii) moving the cast in the queue, and (iv) moving the cast from the position of top of the queue to the position of the outgoing link and destroying it.

Then, we will describe the concrete specification of the animation in real-time LOTOS by using the primitives in Table 3. Here, in order to describe the animation of the above (ii) simply, we define a sub process $Qent$, which represents an entry of each queue and displays an animation to move a cast from its left side entry to its right side entry. $Qent$ can be described as follows.

```
process Qent[v_q_in,v_q_out](wid,interval,pos_x,pos_y):noexit:=
  hide dummy in
  loop
    v_q_in?cid:cast_t;
    dummy!MoveCast(wid,cid,pos_x,pos_y,interval);
    wait(interval);
    v_q_out!cid;
  endloop
endproc
```

Here, variable wid indicates the window ID in which casts are displayed. $Qent$ receives a cast ID (cid) from gate v_q_in, moves the cast to a position (pos_x, pos_y) in time specified by $interval$, and sends cid to the gate v_q_out. Since the queue has eight entries, the whole animation of the queue can be composed of eight $Qent$s. Animations for the above (i) and (iii) can be described similarly. The whole visualization scenario for (i), (ii) and (iii) is described as follows.

```
process V_Queue[a,b]
  (i,wid,cast,width,interval,in_x,in_y,q_x,q_y,width:int):noexit:=
  hide q0,q1,q2,q3,q4,q5,q6,q7,q8,dummy in
   (
    loop
      var cid:cast_t in
        a?class?pct!OK[class=i];
        ?cid:=CreateCast(wid,cast,in_x,in_y);
        q0!cid
      endvar
    endloop
   )
   |||
     Qent[q0,q1](interval,pos_x,pos_y)
   |[q1]|
     Qent[q1,q2](interval,pos_x+width,pos_y)
   |[q2]|...
     Qent[q0,q8](interval,pos_x+width*7,pos_y)
   |[q8]|
   (
    loop
      b?class?pct!OK[class=i];
      q8?cid:cast_t;
      dummy!DestroyCast(wid,cid);
    endloop
   )
endproc
```

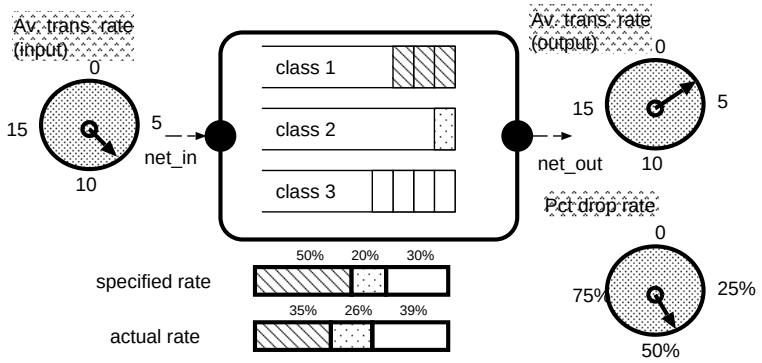


Figure 8: Visualization of protocol performance

Here, we define the position where the packets come from as (in_x, in_y), the position of the entrance of the queue as (q_x, q_y), the width of each entry of the queue as $width$, and the time for moving casts as $interval$.

Other animations for (1), (3) and (4) can be described as processes V_In, V_Drop and V_Out similarly to V_Queue. Consequently, the whole visualization scenario is described as follows.

```
process VS[net_in,a,b,net_out](net_in_x,net_in_y,drop_x,drop_y
                     q_x,q_y,net_out_x,net_out_y,q_width,q_height,
                     cast1,cast2,cast3,interval,cast1,cast2,cast3):noexit=
  V_In[net_in,a](net_in_x,net_in_y,cast1,cast2,cast3)
 |||
  V_Drop[net_in,a](drop_x,drop_y,cast1,cast2,cast3)
 |||
  V_Queue[a,b](1,cast1,width,interval,net_in_x,net_in_y,q_x,q_y,width)
 |||
  V_Queue[a,b](2,cast2,width,interval,net_in_x,net_in_y,q_x,q_y+height,width)
 |||
  V_Queue[a,b](3,cast3,width,interval,net_in_x,net_in_y,q_x,q_y+height*2,width)
 |||
  V_Out[b,net_out](net_out_x,net_out_y,cast1,cast2,cast3)
endproc
```

4.3 Visualization scenario for performance monitoring

In the previous section, we have animated how DRR algorithm works. However, sometimes we would like to monitor the performance of protocols in real-time by animations.

So, as shown in Fig. 8, we describe another scenario to visualize average transmission rates at net_in and net_out, actual proportion among classes in total bytes of packets processed, ratio of dropped packets, and so on.

Average transmission rate at net_in can be calculated by $count/P$ where P is the monitoring interval and $count$ is increased by size of the packet when $net\_in?packet$ is executed in DRR. We show an example description below.

```
process AvRate[net_in, r]:noexit:=
```

```
  loop
    var count:int:=0 in
      loop
        net_in?packet; ?count:= count+size(packet)
      endloop
    [> wait(P); r!count*8/P; ?count:=0;
    endvar
  endloop
endproc
```

In a visualization scenario for the above process, it just shows the transmission rate visually on the $Panel$ when it receives the rate through gate r.

```
process SpeedMeter[r]:noexit:=
  loop
    r?speed:int; UpdateMeter(speed)
  endloop
endproc
```

(Here, $UpdateMeter$ is a process which updates the value of $speed$ to the meter visually where its behavior expression is omitted).

Next, we visualize actual proportion of classes in total bytes of processed packets. We can obtain the size of a packet processed at each class queue by event $b?class?packet!OK$ of DRR.

```
process ClassRate[b, r]:noexit:=
  loop
    var outbytes:list:={0,0,...,0} in
      loop
        b?class?packet!OK; ?outbytes:= Add(outbytes,class,size(packet))
      endloop
    [> wait(P); r!outbytes; ?outbytes:={0,...,0}
    endvar
  endloop
endproc
```

(Here, $outbytes$ is a list variable with $MaxClass$ items. $Add(x, i, n)$ is a function which adds n to i-th item of x.)

In a visualization scenario for process $ClassRate$, it receives the value of $outbytes$ and shows it graphically as a bar chart. An example scenario can be described as follows.

```
process BarChart[r]:noexit:=
  loop
    r?outbytes:list; UpdateBarChart(outbytes)
  endloop
endproc
```

(Here, $UpdateBarChart$ is a process which reflects the value of $outbytes$ to a graphical window as a bar chart as shown in Fig. 8.)

According to the above discussion, the whole visualization scenario is described as follows.

```
process VS[net_in,net_out,b]:noexit:=
  hide r1,r2,r3 in
    (AvRate[b,r1] |[r1]| SpeedMeter[r1])
||| (AvRate[net_out,r2] |[r2]| SpeedMeter[r2])
||| (ClassRate[b,r3] |[r3]| BarChart[r3])
endproc
```

(Here, initializations of casts are omitted)

The above scenario can be used by just replacing the scenario explained in the previous section.

In the proposed method, since an original specification and a visualization scenario are described as separate processes, we can easily change scenarios depending on situations although several scenarios have to be prepared in advance. Moreover, as explained in Sect. 2, we can dynamically change among these scenarios when each of specific events is executed.

5 Experimental results

In order to show the effectiveness of the proposed visualization method, we have carried out some examinations about (1) the performance of our animation library, (2) the performance of our pseudo EDF scheduling mechanism, and (3) the performance of multi-way synchronization in the Java programs generated by our real-time LOTOS compiler. We have used a Pentium III 800MHz PC with 512MB memory for the experiment.

For evaluation of (1), we have examined how many casts can be animated smoothly by our animation library. We have described an example specification such that many casts of 20×36 pixels (some of the pixels are translucent) are animated on the background image that is a JPEG picture of 640×480 pixels. We have examined how frequently the animation window can be updated in every second as varying the number of casts to be displayed. When the number of casts is less than 10, the animation window was updated at about 60 frames per second. In the case of 100 casts, the window was updated at about 30 frames per second. We think this result shows that our animation library is powerful enough for protocol animation.

Next, for evaluation of (2), we have calculated the overhead of our pseudo EDF scheduling algorithm. For this examination, we have described an example specification in which 200 threads are defined such that the starting time of each thread $n(0 \leq n < 200)$ is $100 \times n$ msec and the deadline is 100 msec after the starting time. Here, each thread executes a task taking a specific time to complete. It took about 48 msec to sort 200 threads in increasing order of deadlines. We think this overhead is allowable. For the case that the threads execute a task taking 80 msec, the starting time and deadline of all threads are kept correctly. This shows that the threads can use 80 % of the CPU time for actual processing in our pseudo EDF scheduling mechanism.

Then for evaluation of (3), we have described a simple specification such that some threads continuously execute multi-way synchronization. We have calculated the overhead of multi-way synchronization with changing the number of synchronizing threads. First, we define the threads simply as executing one event continuously. The result is shown in Table 7. Next, in the case of two threads where each of them has three candidates of events to synchronize, it took about 134 msec to finish execution of synchronization 1000 times. Although it takes more time for synchronization when we specify time constraints due to overhead of the EDF mechanism, we think this result is practical enough for development of actual protocols with animations.

Table 7: Consumed time to execute multi-way synchronization for 1000 times

#Threads	1	2	3	4	5	6
time(msec)	99	166	236	305	404	502

6 Conclusion

In this paper, we have proposed a method for animating protocols based on event-driven visualization scenarios described in real-time LOTOS. Main characteristics of our visualization method are that (1) the dynamic visualization using several different scenarios becomes possible depending on the situation while the given specification is being executed, 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. Although we supposed that given original specifications are described in real-time LOTOS throughout the paper, if we can capture events through network or systems, we can apply our method to animate real-life protocols without their source programs. Moreover, our real-time LOTOS compiler generates Java programs in Applet format. So, the proposed method and our compiler can be applied to development of web-based network monitoring systems.

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 that the several parallel processes can execute the 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 want to synchronize the end point of the event, the original specification need be modified.

In our tools, currently we must describe each visualization scenario in real-time LOTOS. To compose the visualization scenario more easily, some tools for helping designers to develop animation behaviors are essential. There will be another approach to automatically generate prototype animations from a given specification. Part of our future work is to develop such tools.

Acknowledgement

The authors would like to thank Mr. Takao Kamon and Ms. Chiyoko Moriyama, the students of Department of Informatics and Mathematical Sciences, Osaka University, for their cooperations to complete experiments.

References

[1] Aaltonen, T., Katara, M. and Pitkanen, R.: “DisCo Toolset - The New Generation”, *Journal of Universal Computer Science*, Vol. 7, No. 1, pp. 3–18 (2001).

[2] Amer, P. D. and New, D.: “Protocol Visualization in Estelle”, *Computer Networks and ISDN Systems*, Vol. 25, pp. 741–760 (1993).

[3] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z. and Weiss, W. : “An Architecture for Differentiated Services”, *IETF RFC2475* (1998).

[4] Braune, B. and Wilhelm, R.: “Focusing in Algorithm Explanation”, *IEEE Trans. on Visualization and Computer Graphics*, Vol. 6, No. 1, pp. 1–7 (2000).

[5] Chen, V., Richter, C., Graf, M. and Brumfield, J.: “VERDI: A Visual Environment for Designing Distributed Systems”, *Journal of Parallel and Distributed Computing*, Vol. 8, No. 6, pp. 128–137 (1990).

[6] S. Floyd, and V. Jacobson: “Link-Sharing and Resource Management Models for Packet Networks”, *IEEE/ACM Trans. on Networking*, Vol. 3, No. 4 (1995).

[7] Garavel, H. and Sighireanu, M.: “French-Romanian integrated proposal for the user language of E-LOTOS”, *Input Document (KC3) to ISO/IEC JTC1/SC21/WG7/ 1.21.20.2.3 ‘Enhancements to LOTOS’* (1996).

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

[9] ISO/IEC: “Information Technology - E-LOTOS”, *ISO/IEC DIS 15437* (2001).

[10] Katagiri, H., Yasumoto, K., Kitajima, A., Higashino, T. and Taniguchi, K.: “Hardware Implementation of Communication Protocols Modeled by Concurrent EFSMs with Multi-Way Synchronization”, *Proc. of 37th Design Automation Conf. (DAC’2000)*, pp. 762–767 (2000).

[11] Kraemer, E. and Stasko, J. T. : “Creating an Accurate Portrayal of Concurrent Executions”, *IEEE Concurrency*, pp 36-46 (1998).

[12] Kraemer, E. and Stasko, J. T. : “The Visualization of Parallel Systems: An Overview”, *Journal of Parallel and Distributed Computing*, Vol. 18, No. 6, pp 36–46 (1993).

[13] Leonard, L. and Leduc, G.: “An introduction to ET-LOTOS for the description of time-sensitive systems”, *Computer Networks and ISDN Systems*, Vol. 29, No. 3, pp. 271–292 (1997).

[14] Leonard, L. and Leduc, G.: “A formal definition of time in LOTOS”, *Formal Aspects of Computing*, Vol. 10, pp. 248–266 (1998).

[15] Parekh, A. and Gallager, R.: “Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case”, *IEEE/ACM Trans. on Networking*, Vol. 1, No. 3, pp. 344–357 (1993).

[16] Shreedhar, M. and Varghese, G.: “Efficient Fair Queuing using Deficit Round Robin”, *Proc. of ACM SIGCOMM’95*, pp. 231–242 (1995).

[17] Stasko, J.T.: “Animating Algorithms with Xtango”, *SIGACT News*, Vol. 23, No. 2, pp. 67–71 (1992).

[18] Turner, K. J. and McClenaghan, A.: “Visual Animation of LOTOS using SOLVE”, *Proc. of IFIP 7th Intl. Conf. on Formal Description Techniques (FORTE’94)*, pp. 283-285 (1994).

[19] Vissers, C. A., Scollo, G., Sinderen, M. v. and Brinksma, E.: “Specification Styles in Distributed Systems Design and Verification”, *Theoretical Computer Science*, Vol. 89, No. 1, pp. 179 – 206 (1991).

[20] Yasumoto, K. and Kitajima, A. and Higashino, T. and Taniguchi, K. : “Hardware Synthesis from Protocol Specifications in LOTOS”, *Proc. Joint Int. Conf. on 11th Formal Description Techniques and 18th Protocol Specification, Testing, and Verification (FORTE/PSTV’98)*, pp. 405–420 (1998).

[21] Yasumoto, K., Higashino, T. and Taniguchi, K. : “A compiler to implement LOTOS specifications in distributed environments”, *Computer Networks*, Vol. 36, No. 2–3, pp. 291–310 (2001).