<table>
<thead>
<tr>
<th>Title</th>
<th>Reflective Specification: Applying A Reflective Language To Formal Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author</td>
<td>MOTOSHI Saeki, Takeshi Hiroi, Takanori Ugai</td>
</tr>
<tr>
<td>Issue Date</td>
<td>1993, 12</td>
</tr>
<tr>
<td>Copyright</td>
<td>(c)1993 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.</td>
</tr>
</tbody>
</table>
Reflective Specification: Applying A Reflective Language To Formal Specification

Motoshi Saeki† Takeshi Hiroi† Takanori Ugai‡
†Dept. of Electrical & Electronic Engineering, Tokyo Institute of Technology
Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
E-mail: {saeki,hiroi}@cs.titech.ac.jp
‡Institute for Social Information Science, Fujitsu Laboratories LTD.
Nakase 1-9-3, Mihama-ku, Chiba-shi, Chiba 261 Japan
E-mail: ugai@iias.flab.fujitsu.co.jp

Abstract

This paper reports on a technique for specifying concurrent systems by using a formal specification language with reflective computation mechanism. We call the specifications written by a reflective language reflective specifications. Our reflective language is an enhanced version of LOTOS (Language of Temporal Ordering Specification). We embedded reflection or reflective computation facilities to behaviour specification part of LOTOS in order to define complex behaviour in simple and natural way. Reflection in a program is a mechanism to access and modify its execution states which its executor has. This mechanism allows us to change dynamically the computational semantics of programs during their execution. Reflection mechanism provides expressive power and flexible enhancement for programming languages. In general, reflective facilities cause to introduce two level descriptions of programs — object level descriptions and meta-level descriptions. The object level descriptions are considered as data of its meta-level descriptions. As well as programs, the reflection allows us to construct comprehensive specifications of complex systems.

Specifications written in LOTOS consist of behaviour specifications which define the observable interaction sequences of the systems, and data specifications. We have introduced reflection facilities to behaviour specification parts in order to define complex behaviour in simple and natural way. Our enhanced version of LOTOS is called RLOTOS, and has two level architecture — object level and meta level. The processes in the meta level, called meta processes, have the computational information and interpret the behaviour expressions of their object level processes. We can define meta processes in the same manner as LOTOS processes to control the behaviour of the object level processes. In this paper, we present a case study of specifying an operating system by using RLOTOS. Furthermore we discuss the method to construct comprehensive formal specifications by using reflective languages and explore the applicability of the reflective language to formal specification. The essential point of comprehensiveness is that the meta properties of the system such as control characteristics can be specified separately from the object level properties.

1 Introduction

LOTOS (Language of Temporal Ordering Specification) [2] which has been developed for formal specification of communication systems, has the powerful constructs for describing concurrency, non-determinism, synchronous and asynchronous interaction, and interruption. LOTOS has been standardized in International Organization for Standardization (ISO), and its many practical application examples have been reported.

Reflection or reflective computation[10, 12] in a program is a mechanism to access and modify its execution states which its executor has. This mechanism allows us to change dynamically the computational semantics of programs during their execution. Reflection mechanism provides expressive power and flexible enhancement for programming languages. In general, reflective facilities cause to introduce two level descriptions of programs — object level descriptions and meta-level descriptions. The object level descriptions are considered as data of its meta-level descriptions. As well as programs, the reflection allows us to construct comprehensive specifications of complex systems.

Specifications written in LOTOS consist of behaviour specifications which define the observable interaction sequences of the systems, and data specifications. We have introduced reflection facilities to behaviour specification parts in order to define complex behaviour in simple and natural way. Our enhanced version of LOTOS is called RLOTOS (Reflective LOTOS). In this paper, we discuss the benefit of applying the reflective languages to formal specifications by using RLOTOS. We call the specifications written in a reflective language reflective specifications. This paper is structured as follows. Section 2 and section 3 are introductory sections for LOTOS and for RLOTOS respectively. In section 4, we show an example of the specification written in RLOTOS — a MINIX operating system. Furthermore we discuss the method to construct comprehensive formal specifications by using reflective languages and investigate applicability of the reflective language to formal specification in section 5. The essential point of comprehensiveness is that the meta properties of the system such as control characteristics can be specified separately from the object level properties.

2 LOTOS

The descriptions written in LOTOS consist of behaviour specifications and data specifications. The formal semantics of LOTOS is based on CCS[6] for behaviour specification and on Algebra of Abstract Data Type(ADT) for data specification. The behaviour
Constructors  | Naming  | Intuitive Meaning
--- | --- | ---
$a; B$  | action prefix  | The event $a$ occurs, and after that $B$
$B_1||B_2$  | choice  | Either $B_1$ or $B_2$ is executed
$[G_1] \rightarrow B_1$  | choice with guard  | If $G_i$ ($i = 1, 2$) then, $B_i$ is executed
$[G_1][G_2] \rightarrow B_2$  | Interleaving Operator  | $B_1$ and $B_2$ are independently, i.e. asynchronously executed in parallel
$B_1||B_2$  | Synchronizing Operator  | $B_1$ and $B_2$ are executed in parallel and synchronously with all events.
$B_1[[a_1, ..., a_n]]B_2$  | General Parallel Operator  | $B_1$ and $B_2$ are executed in parallel and synchronously with $a_1, ..., a_n$.

| Table 1: Basic Constructor of Behavior Expressions |

specifications define the observable behaviour, i.e. interaction sequences of the system to be specified. The system to be specified is captured as a set of processes communicating with each other at their gates. These processes may be decomposed into several subprocesses hierarchically. The atomic unit of the interaction is an event, and it is also an unit of synchronized interaction. The process description contains the behaviour expression defining the observable behaviour of the process. LOTOS has several constructors for behaviour, e.g. "; (action prefix)" and "\(\Rightarrow\) (enabling operator)" for sequential composition, "[] (choice operator)" for selection, and "[[;]] (general parallel operator)" for parallel composition with synchronous events $a_1, ..., a_n$. The basic constructors for behaviour expressions are listed in Table 1.

Intuitively speaking, the event "$g ?x$ int", where $g$ is a gate name, represents the input of an integer through the gate $g$ and the assignment of the input data to the variable $x$. The event "$g !a$" where "$a$" is a value stands for an output of the value "$a$" through the gate $g$. Abstract data type definitions express what values are handled. Let's consider a small LOTOS specification:

**specification example [input, output]**

```plaintext
: noexit :=
type Integer is
sorts int
opns (0 : Int) s : int → int
endtype

behaviour
hide middle in
(input ?x:int; middle !x : stop)
[[middle]]
(middle ?y:int [not(y = 0)]; output !y : stop)
endspec
```

where "in", "middle", and "out" are gate names. This specification specifies the system as shown in Figure 1. The statements from **type** to **endtype** represent an abstract data type definition of "integer". The terms of the sort "int" are constructed from "0" and the repetitive application of "s", i.e. 0, s(0), s(s(0)), ... are examples of the terms. The statements from **behaviour** to **endspec** are a definition of the behaviour of the buffer.

```
(in ?x:int ; middle !x : stop)
[[middle]]
(middle ?y:int [not(y = 0)]; output !y : stop)
```

is an example of behaviour expression. This behaviour expression shows two kinds of event sequences may occur in parallel. They communicate to each other synchronizing with the gate "middle". The behaviour expression in the left hand side of "[[middle]]" sends synchronously the integer value x, which is an input through the gate "in", to the right hand side through the gate "middle". The right hand side can receive the value only if the value is not 0 because the condition "[not(y = 0)]" is attached. The operator "hide ...\(\Rightarrow\) in" is called hiding operator and it shields off the specified events from the external environment. Thus the event "middle" is not observed from the outside of the buffer. Actually the behaviour expression provides an event sequence "in, out" as an observable behaviour. And it represents the right hand receives the value, and then outputs it through the gate "out". The gate names with variables and/or values such as "in ?x:int" and "out !y" are syntactically called "action-denotation", and semantically denote the events which
include data-passing.

We will show another example.

```
specification example2[in,out] : exit :=
   (* defines for integer *)
   behaviour
   in ?x:int
   [\{x > 5\} \rightarrow \text{out} \text{is}(x) ; \text{exit}]
   [\{x \leq 5\} \rightarrow \text{out} \text{is}(x) ; \text{exit}]
endspec
```

where operations \(\text{\textgreater{}}\) (greater than), \(\text{\textless{}}\) (less than or equal to) are supposed to be defined in the integer module by usual way, and 5 is an abbreviation of \(s(s(s(s(0))))\). This description provides a behaviour or control structure as shown in Figure 2. In this example, the system gets an integer number, if it is greater than 5, the system puts out the number which is added by 1. And if it is less than or equal to 5, the system puts out the number as it is.

3 LOTOS with Reflection — RLOTOS

3.1 Reflection Mechanism in RLOTOS

RLOTOS (Reflective LOTOS) is an enhanced version of LOTOS with reflective mechanism. Similar to other reflective concurrent languages\([15, 11]\), RLOTOS has hierarchical architecture, which includes object level and meta level which is a level for controlling the execution of the object level descriptions. Reflective mechanism has been embedded not to ADT part but to behaviour expression part. A labelled transition system is used to model the execution of behaviour expressions of LOTOS. Let’s consider the behaviour expression \(a; B\) as an example. The transition rule for the expression \(a; B\) is \(a; B \rightarrow B\). From the operational viewpoint, it denotes that \(a; B\) can execute \(a\) and thereby turn into \(B\), i.e. the occurrence of the event \(a\) in state \(a; B\) leads us into the state \(B\).

The essential information of this transition rules is a current state \((a; B)\) and a pair of the next possible event \((a)\) and the next state \((B)\). Thus we can change and control the transition by providing the next events and the next states explicitly. A behaviour expression described in the meta level provides a next event and a next state for the execution of the behaviour expressions in the object level.

The behaviour expressions in the meta level can be constructed by the same constructors as the object level, which are shown in Table 1. Figure 3 shows how the meta level of RLOTOS interprets the object level. In the case of the original LOTOS, it can be considered that the only one meta process interpreter interprets the behaviour expressions of the object level processes and executes them following the transition rules. In the interpreter, the LOTOS source codes in object level or descriptions are handled as data themselves. Since terms in abstract data types express data, we should establish the method to define LOTOS descriptions with the terms. As discussed in \([7]\), behaviour expressions and other syntactic categories of LOTOS can be defined as terms of abstract data types, whose constructors are the operators mentioned the previous section such as action prefix, enabling operator, choice operator and so on. We use the sort names Event and Bexp to express the syntactic categories “action-denotation” and “behaviour expression” respectively. For simplicity, we will use the notation ‘in ?x:int’ to express a term denoting the action-denotation “in ?x:int” in this paper. The sort of ‘in ?x:int’ is Event.

In RLOTOS, we can specify the meta processes and they communicate with the interpreter through the special gates called reflective gates in order to control the execution of the object level. We have three reflective gates — “currentg”, “nextg”, and “controlg”, which are used for referring to information about the execution of the object level description and for changing its behaviour. In addition, we should note that the meta processes can communicate with the object level processes except for reflective gates. However, we do not use these communications to specify the systems by the reason mentioned later.

A behaviour expression to be currently executed is passed through the reflective gate “currentg.” The reflective gate “nextg” outputs a set of pairs of a next possible event in and the expression after the next event occurs, i.e. a next event and a next state. Both of them are represented as ADT terms. If you change the execution of the object level process dynamically during its execution, you send to the gate “controle” a pair of a new event and a new behaviour expression corresponding to the next state. It may be different from one specified by labelled transition rules of LOTOS. You need not specify an interpreter process but the meta processes and the object level processes in RLOTOS. Figure 4 shows how to access and how to control the execution by using reflective gates. The meta process proceeds to the next step of the execution of the object level whenever it outputs to controlg.

3.2 Specifications in RLOTOS

The syntactical structures of the specifications written in RLOTOS is as follows.

```
specification <specification identifier> ...
<data type definitions>
   (* Definition of Abstract Data Types *)
   ...
```

Figure 2: An Example: example2
Figure 3: Relationship between Meta Level and Object Level

(b) Reflective LOTOS (RLOTOS)

Figure 4: Basic Reflective Procedure in RLOTOS

behaviour of object level
<behaviour expression of object level>
(* Specification of behaviour in object level *)

where
<process definitions>
(* processes used in behaviour expression are defined *)

behaviour of meta level
<behaviour expression of meta level>
(* Specification of behaviour in meta level *)

where
<meta process definitions>
(* Similar to object level processes *)

behaviour of meta meta level
(* Similar to meta level *)

end spec

The following example of meta level descriptions provides the execution control which follows the LOTOS transition rules, i.e. the same execution as LOTOS interpreter does.

specification trivial : noexit
(* Definition of ADT *)

behaviour of object level
(* Behavior Expression *)...

behaviour of meta level
trivial_process[nextg,controlg]

where
process trivial_process[nextg,controlg] : noexit :=
nextg?x : EventPairSet ; controlg|choice(x) ;
trivial_process[nextg,controlg]
endproc
endspec

The trivial process obtains a set of pairs of a next possible event and a next behaviour expression through the nextg gate. “EventPairSet” is a sort name of the set of the pairs of an event and a behaviour expression, and is defined in built-in abstract data types of RLOTOS. The operation “choice” on EventPairSet is also one of the built-in operations of ADT. It chooses non-deterministically an element from a set given as an input. The trivial process chooses a pair of an event and an expression of a set “x” and return it through controlg as a next event and a next state.

4 Example — MINIX Operating System

In this section, we introduces a typical example of a specification written in RLOTOS — MINIX operating system. We use the control mechanism of the meta processes to the object processes to specify the example. Control features and the normal behaviour of the example systems are described separately in the meta level and in the object level. To clarify the separation and to construct comprehensive specifications, we have not used the communication mechanism between the object level and the meta level except for reflective gates as shown in Figure 3. This restriction...
MINIX Operating System, UNIX compatible operating system, has been developed by A.W.Tanenbaum for educational purposes [13]. We focus on the process management part of MINIX. MINIX consists of four layers as shown in Figure 5(a). The processes in the layers 2, 3, and 4 are running in (pseudo) parallel, communicating with each other using messages. The lowest level layer, the process manager, provides a model of sequential processes for the higher level layers. It selects a process from the higher layer processes which are waiting for running, and switches rapidly a running process to the selected one. The scheduling mechanism, i.e. the way to select a process for its running, is a priority scheduling using a three-level queuing system. Each level corresponds to the layer 2, 3, and 4 in Figure 5(a), and round robin scheduling is used within it. The processes in the layer 2, i.e. I/O tasks, have the highest priority, the memory manager(MM) and the file server(FS) in the layer 3 are next and the user processes are the lowest priority. A process cannot be running until no processes with the higher priority are ready to run. Thus the memory manager or the file server is selected as a running process if no I/O tasks are ready to run.

Message passing mechanism between MINIX processes is based on rendezvous principle. The processes communicate with each other by using two system calls — send and receive. If the send system call is done before the corresponding receive system call, the sending process is blocked until the receive is done. Similarly, if the receive is done before the send, the receiver waits until the send is executed in the sender process. The process manager maintains the queue ready_q which consists of three sub queues of the process identifiers. Each sub queue holds the runnable (ready to run) processes in each layer as shown in Figure 6. We can have the following description of the abstract data type "Ready_queue", abstract data types in RLOTOS. In this description, several operators for general queues such as "remove" and "first" are used and are defined in [2]. "first(queue)" and "remove(queue)" produce the first element at the end of the queue and the queue after the first element is removed respectively.

**Type** Ready.Queue is QueueOfProcessId

**ops**

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_q</td>
<td>Ready_queue — QueueOfProcessId</td>
</tr>
<tr>
<td>server_q</td>
<td>QueueOfProcessId — Ready_queue</td>
</tr>
<tr>
<td>user_q</td>
<td>QueueOfProcessId — Ready_queue</td>
</tr>
<tr>
<td>pick_process</td>
<td>Ready_queue — ProcessId</td>
</tr>
</tbody>
</table>


**eqns**

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for all pid:ProcessId, tq,sq,qu:QueueOfProcessId</td>
<td></td>
</tr>
<tr>
<td>task_q(&lt;tq,sq,qu&gt;)</td>
<td>= tq ;</td>
</tr>
<tr>
<td>server_q(&lt;tq,sq,qu&gt;)</td>
<td>= sq ;</td>
</tr>
<tr>
<td>is_empty(tq)</td>
<td>⇒</td>
</tr>
<tr>
<td>first(priority_process(&lt;tq,sq,qu&gt;))</td>
<td>= first(tq) ;</td>
</tr>
<tr>
<td>is_empty(sq)</td>
<td>⇒</td>
</tr>
<tr>
<td>is_not_empty(sq)</td>
<td>⇒</td>
</tr>
<tr>
<td>pick_process(&lt;tq,sq,qu&gt;)</td>
<td>= &lt;remove(tq),sq,qu&gt; ;</td>
</tr>
<tr>
<td>is_empty(sq)</td>
<td>⇒</td>
</tr>
<tr>
<td>is_not_empty(sq)</td>
<td>⇒</td>
</tr>
<tr>
<td>pick_process(&lt;tq,sq,qu&gt;)</td>
<td>= &lt;tq,remove(sq),qu&gt; ;</td>
</tr>
<tr>
<td>kind_of_process(pid)</td>
<td>⇒</td>
</tr>
<tr>
<td>put_process(pid,&lt;tq,sq,qu&gt;)</td>
<td>= &lt;add(pid,tq),sq,qu&gt; ;</td>
</tr>
</tbody>
</table>

Let's consider how to describe the process management part of MINIX using RLOTOS. Each process in the layers 2, 3, and 4 is described in a LOTOS process of the object level. The process manager in the layer 1, which schedules and controls the processes, can be defined in the meta level of RLOTOS. The
structure of MINIX in RLOTOS is shown in Figure 5 (b). The independent LOTOS parallel processes are executed in the arbitrary interleaving of their events. A process in the meta level controls the interleaving of the events of the processes in the object level such as the Disk Task, the memory manager, the file server, and the user processes. The meta level process takes off the events in the ready or blocked processes from the candidates of the next possible events, even if they can occur next by LOTOS transition rules. It uses the built-in operator of RLOTOS “process.id”, where “process.id(event)” produces a process identifier which participates in “event”. This identification of the running process’s event can be made by comparing “running-pid”, which holds the running process identifier, with the value of “process.id(event)”. “Timeout” event lets the process manager know the time comes when it should switch the running process. When the timeout occurs, the process manager adds the running process to the ready-q queue and selects the process with the highest priority from the ready-q queue as a new running process. The operation “pick-process” defined in ADT computes the process identifier with highest priority. The process identifier of the new running process is assigned to the running-pid.

Message passing mechanism can be defined in the meta level. The sender process offers a send event, and it corresponds to calling a send system call in MINIX. The process manager checks if the receiver process has been blocked to receive the message, i.e. waiting for arrival of the message. The receive_q queue holds the blocked receiver processes which are waiting for the corresponding send system calls. If the receiver process is in the queue, the sender process continues the send event offer and the receive event occurs in the receiver process successively. After that, the process manager removes the receiver from receive_q and adds it to ready_q. That is to say, the receiver process becomes runnable. In contrast with the above case, if the receiver is not in receive_q, i.e. the corresponding receive system call does not happen yet in it, the sender becomes blocked and is added to send_q.

Both send_q and receive_q have pairs of a sender and a receiver process identifier. The behaviour structure of the meta processes for MINIX’s process management part is shown in Figure 7 and its description by RLOTOS is as follows.

**MINIX Operating System by RLOTOS**

**specification MINIX : noexit**

(* Definition of Abstract Data Types such as Ready_queue *)

**behaviour of object level**

DiskTask || TTYTask || ... || MM || FS ||

Init || UserProcess(0) || UserProcess(a(0)) || ...

**where**

(* Definition of Object Level Processes, i.e. Processes in the Layers 2, 3, and 4 *)

**process DiskTask : noexit :**

(action#1 ; action#2 ; ...**

endproc **

...**

**behaviour of meta level**

process manager[currentg,nextg,controlg,timeout] [InitReady.q,{},{},InitRunningPid]

**where**

process manager[currentg,nextg,controlg]

(ready.q:Ready_queue, send.q, receive.q:QueueOfProcessIdPair, running.pid:ProcessId) : noexit :=

hide timeout in

(* check the next events possible to occur, and control them *)

(currentg?currentexp:Bexp ; nextg?nextset:EventPairSet ;

process_manager[currentg,nextg,controlg]

(choice(nextset),nextset,ready.q,send.q, receive.q, running.pid)

(* choice(nextset) denotes a candidate for an next event and a next expression. *)

[]

(* process switch by the timeout event *)

timeout ;

process manager[currentg,nextg,controlg]
(pick_process(put_process((running.pid, ready.q)), send.q, receive.q),
first.priority.process
(put_process((running.pid, ready.q))))

where

process process.manager1[currentg, nextg, controlg]
(nexpair:EventPair, nextset:EventPairSet,
ready.q, ready.q, queue,
send.q, receive.q, QueueOfProcessIdPair,
running.pid:ProcessId)

:noexit :=
let nextevent:Event = event_part(nextpair),
nextbexp:Bexp = expression_part(nextpair) In
[process_id(nextevent) = running.pid] \rightarrow
(* The possible event participates
in the running process *)

[(nextevent = "send!sender!receiver!message") \rightarrow
(* The selected next event is a send system call *)

\rightarrow
(* The receiver process has already been blocked *)

\rightarrow
(* The receiver process is not blocked
i.e. its execution does not reach
the corresponding receive system call yet *)

process.manager1[currentg, nextg, controlg]
(put_process(receiver, ready.q),
send.q, remove(pair(sender, receiver), recipient.q),
running.pid)

(* The next event occurs and the corresponding receive event will occur successively.
(* The blocked receiver is removed from receive.q *)

[[not(pair(sender, receiver) \in receive.q)] \rightarrow
(* The receiver process is not blocked
i.e. its execution does not reach
the corresponding receive system call yet *)

process.manager1[currentg, nextg, controlg]
(choose(remove(nextpair, nextset),
remove(nextpair, nextset),
pick_process(ready.q),
add(pai(sender, receiver), send.q),
receive.q, first.priority.process(ready.q))

(* The sender process is added to send.q
as a blocked process *)


nextevent = "receive!sender!receiver?message" \rightarrow
(* The selected next event is a receive system call *)

...]

[[not(systemcall(nextevent) = "send")
and not(systemcall(nextevent) = "receive")] \rightarrow
(* The selected next event is neither "send"
nor "receive" *)

control\n
process.manager1[currentg, nextg, controlg]
(ready.q, send.q, receive.q, running.id)

(* The next event occurs. *)

[[not(process_id(nextevent) = running.pid)] \rightarrow
(* The selected next event does not participate in the running process *)

process.manager1[currentg, nextg, controlg]
(choose(remove(nextpair, nextset),
remove(nextpair, nextset)), ready.q,
send.q, receive.q, running.pid)

(* The next event does not occur, and choose
another event again *)

endproc
endproc
endspec

Note that the descriptions of the object level processes do not contain the interactions with the process manager but only essential actions for their tasks. Suppose that the DiskTask performs the event sequence "action\#1, action\#2, ... " to achieve its task, i.e. disk drive control. We specify nothing but this event sequence for the DiskTask process.

5 Discussion

In the previous section, we introduced a typical example of the reflective specification, i.e. specification written in the reflective language. First, we will discuss its difference from the specification written in non-reflective language. The next problem is which kind of system we can apply a reflective language effectively to, and how to construct a reflective specification. We will also discuss the considerable method for constructing reflective specification.

5.1 Reflective Specification and Non-reflective Specification

To clarify the characteristics of reflective specification, we introduce the MINIX specification written in original LOTOS language, which is non-reflective.

MINIX Specification by LOTOS

specification MINIX : noexit

(* Definition of Abstract Data Types *)

behaviour

hide port in

(DiskTask[port] || TTYTask[port] || ... || MM[port] ||
FS[port] || Init[port] ||
UserProcess[port](0) || UserProcess[port] (s(0)) ||... ||
[port])

process.manager[port]

(InitReady.q, {}, InitRunningPid)

where

(* Definition of Processes in Section 2, 3, 4 *)

process DiskTask[port] : noexit :=

execution.request[port][DiskTask, 'action\#1'] >>
action\#1;
port!DiskTask!terminated;
execution.request[port][DiskTask, 'action\#2'] >>
action\#2;
port!DiskTask!terminated;
...

endproc

... process execution.request[port]

210
(pid:ProcessId, event:Event): exit :=
  port!pid!event!request ;
  port!pid?answer:Answer ;
  (answer = ack) → exit
  (* The request is granted *)
  | answer = nack → execute_request[port](pid, event)
  (* The request is pending *)
endproc

(* Definition of Process Management Part *)
process process_manager[port]
  (ready.q:Ready_queue,
   send.q, receive.q:QueueOfProcessIdPair,
   running.pid:ProcessId) : noexit :=
  hide timeout in
  (* check the next events possible to occur, and control them *)
  (port?pid:ProcessId?event:Event!request;
   process_manager1[port]
    (pid, event, ready.q, send.q, receive.q, running.pid)
    (* process switch by the timeout event * )
    timeout;
   process_manager[port]
    (pick_process(put_process(running.pid,ready.q)),
     send.q, receive.q,
     first_priority_process
      (put_process(running.pid,ready.q))))
  )

where

process process_manager1[port]
  (pid:ProcessId, event:Event, ready.q:Ready_queue,
   send.q, receive.q:QueueOfProcessIdPair,
   running.id:ProcessId) : noexit :=
  (pid = running.pid) →
  (* The possible event participates in the running process *)
  (event = 'send!sender!receiver!message') →
  (* The requested event is a send system call *)
  (pair(sender,receiver) ∈ receive.q) →
  (* The receiver process has already blocked *)
  port!pid!ack;
  port!receiver!ack;
  (* The corresponding receive event occurs successively *)
  port!pid!terminated;
  port!receiver!terminated;
  process_manager[port]
    (put_process(receiver,ready.q), send.q,
     remove(pair(sender,receiver),receive.q),
     running.pid)
    (* not(pair(sender,receiver) ∈ receive.q) →
     ...
     event = 'receive!receiver!sender!x:message' →
     ...
     not(event_part(event) = send)
and not(event_part(event) = receive)] →
( * The requested event is neither 'send'
  nor 'receive' *)
port.pid!ack;
( * The requested event occurs *)
port.pid!terminated;
process.manager[port]
(ready.q, send.q, receive.q, running.pid)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
)
processes and meta processes should not be employed in our technique results in comprehensive specifications, and seems to be important for applying the reflective languages to formal specification.

5.2 How to Construct Reflective Specification

Next, we will discuss how to construct a reflective specification. The first thing we should do to specify a system is to identify meta properties. How to identify them depends on specification methods for reflective languages. Scheduling, fault recovery, exception handling, and monitoring can be considered as meta features, so reflective languages are useful to specify the systems with these facilities. We also have experienced specifying several kinds of system—a fault tolerant system[4], a lift control system[1], communication protocols, software development process[6], and an interpreter for LOTOS-T (LOTOS with time concept)[3]. The way to separate the meta level and the object level depends on the domain of the systems to be specified, so we can construct specification patterns or styles according to the domain of the systems. These patterns help us to construct the reflective specifications. Four specification styles—monolithic, state-oriented, constraint-oriented, resource-oriented styles—only for original LOTOS has been proposed[14]. We can also extend these styles to ones suitable for reflective languages.

Object oriented analysis[9] can be effectively applied to construction of the reflective specifications, especially identification of the meta properties included in the systems. In object oriented paradigm, information is encapsulated into objects. Information encapsulation may result in the complicated communications for the objects to obtain the global information. Suppose that the lift control system such as discussed in [1]. Each lift always communicates with a scheduler, which holds the state of all lifts, to decide where it should go next. When we depict its object interaction diagram[9], many communications is centralized to the scheduler object. Such objects can be considered as meta level objects. Thus we focus on the objects to which communications are centralized. The same situation has appeared in the MINIX example as shown in Figure 8. This method for identifying the meta objects can be embedded to OOA and we can have reflective object oriented method.

Supporting tools for reflective specifications should provide separate supports for object level descriptions and for meta level descriptions. It is a further research as well as the efficient execution of the reflective descriptions such as [5].

Acknowledgements

The authors wish to thank Prof. C. A. Vissers and Mr. L. F. Pires of University of Twente for their discussions and comments to RLOTOS.

References


