T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

Title	Reflective Specification: Applying A Reflective Language To Formal Specification
Author	MOTOSHI Saeki, Takeshi Hiroi, Takanori Ugai
Journal/Book name	Proc. 7th International Workshop on Software Specification & Design, Vol. , No. , pp. 204-213
発行日 / Issue date	1993, 12
権利情報 / Copyright	(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.

Powered by T2R2 (Science Tokyo Research Repository)

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. 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, nondeterminism, 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

1063-6765/93 $3.00 © 1993 IEEE

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
$[G_1] \rightarrow B_1$ $[][G_2] \rightarrow B_2$	choice with guard	If G_i $(i = 1, 2)$ then, B_i is executed
$B_1 \|\|\| B_2$	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.
$B_1 \|[a_1, \ldots, a_n]\| B_2$	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 ">> (*enabling operator*)" for sequential composition, "[] (*choice operator*)" for selection, and "$|[a_1, ..., a_n]|$(*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]
        : 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

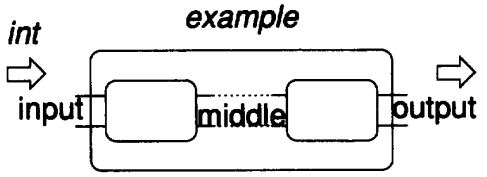


Figure 1: An Example

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)] ; out !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** ··· **in**" is called *hiding operator* and it shileds 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

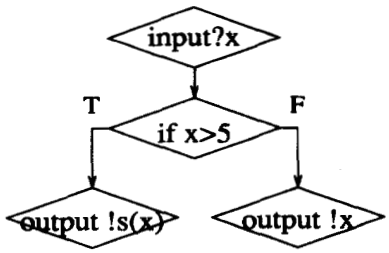


Figure 2: An Example:example2

include data-passing.

We will show another example.

```
specification example2[in,out] : exit :=
(* defines for integer *)
behaviour
 in ?x:int
  ([x > 5] → out !s(x) ; exit
  []
  [x ≤ 5] → out !x ; exit)
endspec
```

where operations "> (greater than)","≤ (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(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 $a; B \xrightarrow{a} B$. 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], behavior 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 uses the sort names Event and Bexp to express the syntactic categories "action-denotation" and "behavior 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.

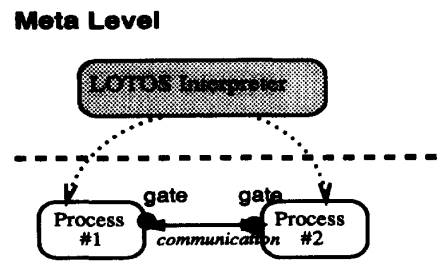
In RLOTOS, we can specify the meta processes and they communicate with the interpreter through the special gates called *reflective gates* in order to control the execution of the object level. We have three reflective gates — "currentg", "nextg", and "controlg", which are used for referring to information 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 "controlg" 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.

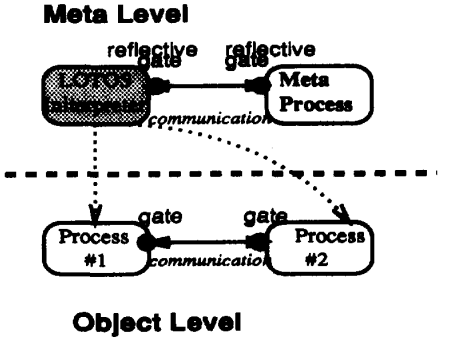
3.2 Specifications in RLOTOS

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

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



(a) LOTOS



(b) Reflective LOTOS (RLOTOS)

Figure 3: Relationship between Meta Level and Object Level

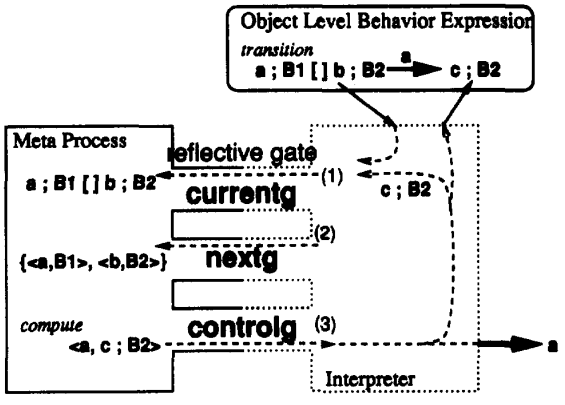


Figure 4: Basic Reflective Procedure in RLOTOS

```
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
```

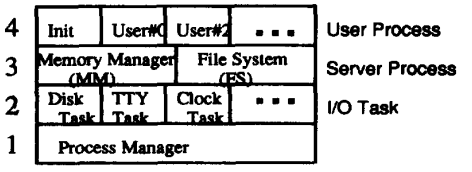
The following example of meta level descriptions provides the execution control which follows the LOTOS transition rules, i.e. the same execution as LOTOS interpreter does.

```
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
```

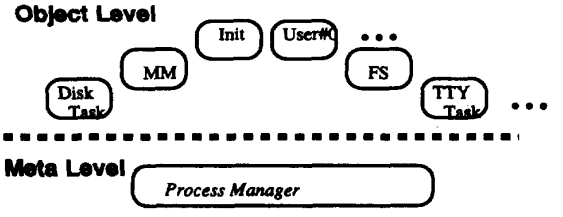
The trivial process obtains a set of pairs of a next possible event and a next behaviour expression through the nextg gate. "EventPairSet" is a sort name of the set of the pairs of an event and a behaviour expression, and is defined in built-in abstract data types of RLOTOS. The operation "choice" on 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 from the 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



(a) MINIX Internal Structure



(b) MINIX Structure from Reflective View

Figure 5: Structure of MINIX

will be discussed in the next section. Figure 8 shows the structural difference between the reflective specifications and the usual specifications, and helps readers understand the reflective specification of the example.

MINIX Operating System, UNIX compatible operating system, has been developed by A.W.Tanenbaum for educational purpose[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
 sorts Ready_queue
 opns
  <_ _ _>
   : QueueOfProcessId, QueueOfProcessId,
     QueueOfProcessId → Ready_queue
  task_q, server_q, user_q
   : Ready_queue → QueueOfProcessId
  pick_process : Ready_queue → Ready_queue
   (* remove a process with the highest priority
      from Ready_queue *)
  first_priority_process : Ready_queue → ProcessId
   (* return the process identifier with
     the highest priority in Ready_queue *)
  put_process
   : ProcessId, Ready_queue → Ready_queue
   (* put a process ProcessId to the Ready_queue *)
 ...
 eqns forall pid:ProcessId, tq,sq,uq:QueueOfProcessId
  ofsort QueueOfProcessId
   task_q(<tq,sq,uq>) = tq ;
   server_q(<tq,sq,uq>) = sq ;
   ...
  ofsort ProcessId
   is_not_empty(tq) ⇒
    first_priority_process(<tq,sq,uq>) = first(tq) ;
   ...
  ofsort Ready_queue
   is_not_empty(tq) ⇒
    pick_process(<tq,sq,uq>) = <remove(tq),sq,uq> ;
   is_empty(tq) and is_not_empty(sq) ⇒
    pick_process(<tq,sq,uq>) = <tq,remove(sq),uq> ;
   ...
   kind_of_process(pid) = task ⇒
    put_process(pid,<tq,sq,uq>)
     = <add(pid,tq),sq,uq> ;
 ...
endtype
```

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

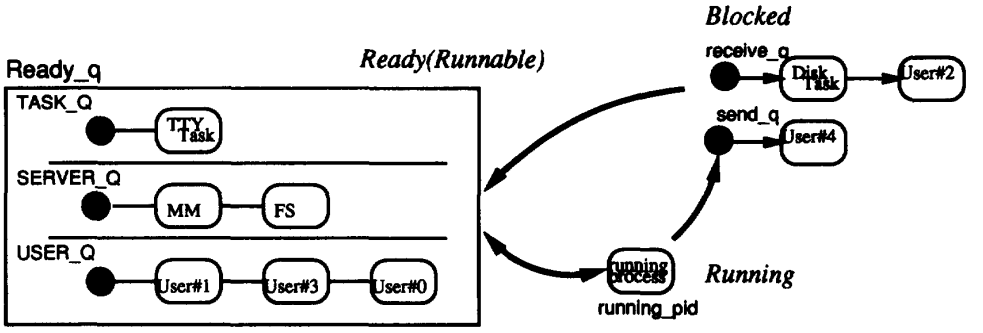


Figure 6: MINIX Process Management

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 ruunable. 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(s(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 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_manager1[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]
  (nextpair:EventPair, nextset:EventPairSet,
  ready_q:Ready_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] →
   (* The possible event participates
   in the running process *)
    ([nextevent ='send!sender!receiver!message'] →
    (* The selected next event is a send system call *)
     ([pair(sender,receiver) ∈ receive_q] →
     (* The receiver process has already been blocked *)
     controlg!pair(nextevent,
      'receive!receiver!sender!message ; nextbexp') ;
     process_manager[currentg,nextg,controlg]
      (put_process(receiver, ready_q),
      send_q, remove(pair(sender,receiver),receiver_q),
      running_pid)
      (* The nextevent occurs and the corresponding
      receive event will occur successively.
      (* The blocked receiver is removed
      from receive_q *)
    []
     [not(pair(sender,receiver) ∈ receive_q)] →
      (* The receiver process is not blocked,
      i.e. its execution does not reach
      the corresponding receive system call yet *)
      process_manager1[currentg,nextg,controlg]
       (choice(remove(nextpair,nextset),
          remove(nextpair,nextset),
        pick_process(ready_q),
        add(pair(sender,receiver),send_q),
        receive_q, first_priority_process(ready_q))
      (* The sender process is add to send_q
      as a blocked process *)
    )
    []
    [nextevent='receive!sender!receiver?x:message'] →
     (* The selected nextevent is a receive system call *)
      ...
    []
    [not(systemcall(nextevent)='send')
     and not(systemcall(nextevent)='receive')] →
    (* The selected nextevent is neither 'send'
    nor 'receive' *)
     controlg!nextpair ;
     process_manager[currentg,nextg,controlg]
      (ready_q,send_q,receive_q,running_id)
     (* The nextevent occurs. *)
  )
 []
 [not(process_id(nextevent)=running_pid)] →
   (* The selected nextevent does not
   participate in the running process *)
    process_manager1[currentg,nextg,controlg]
     (choice(remove(nextpair,nextset)),
     remove(nextpair,nextset), ready_q,
     send_q, receive_q, running_pid)
     (* The nextevent 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 Layer 2, 3, 1nd 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]
```

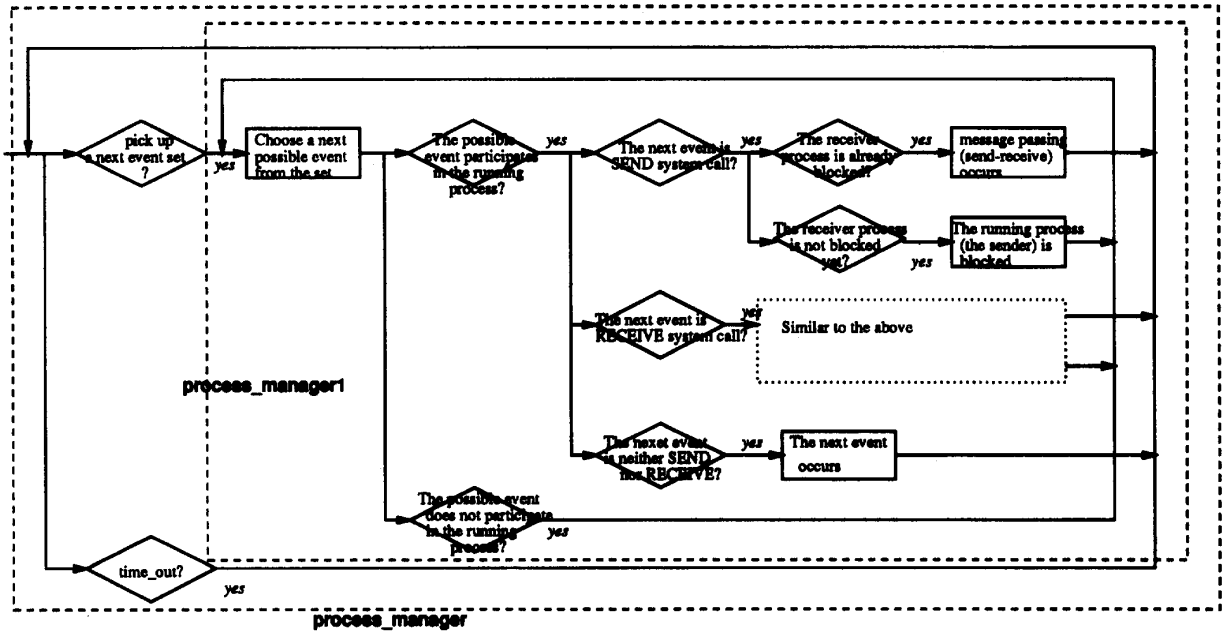
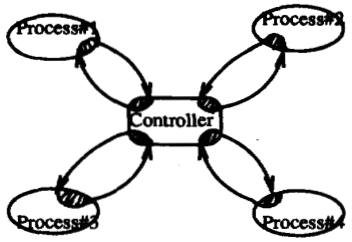


Figure 7: Behaviour Structure of MINIX's Process Management Part in Meta Level

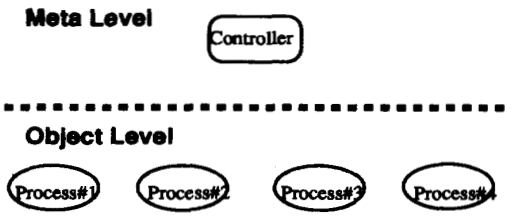
```
  (pid:ProcessId, event:Event): exit :=
  port!pid!event!request ;
  port!pid?answer:Answer ;
  ([answer = ack] → exit
  (* The request is granted *)
  []
  [answer = nack] → execution_request[port](pid, event)
  (* The request is pending *)
  )
endproc
(* Definition of Process Management Part *)
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)
      )
    []
    [not(pid = running_pid)] →
    (* The requested event does not
    participate in the running process. *)
      port!pid!nack ;
      (* The requested event does not occur, and choose
      another requested event *)
      port?newpid:ProcessId
          ?newevent:Event[not(newpid=pid)] ;
      process_manager1[port]
        (newpid, newevent, ready_q, send_q, receive_q,
        running_pid)
    )
  endproc
endproc
endspec
```

We should note that the above code includes the notation which cannot be used in original LOTOS syntax for readability. Its behaviour structure is very similar to the RLOTOS version mentioned in the previous section. Sans serif type style in the above code stands for the LOTOS statements different from the RLOTOS description. We have to introduce a new process called "execution_request". This process receives the requests for executing an event from the processes in the layers 2, 3, or 4, and then asks the process_manager if the requested event can be executed or not. Suppose that the DiskTask process will execute the event sequence "action#1, action#2, ···" for disk drive control. These events in the sequence are controlled by the process_manager. Thus the DiskTask process requests the grant for the execution of action#1 before starting its execution by instantiating the execution_request process. The process communicates with the process_manager through the gate "port". If the execution is granted, the process_manager responds with "ack" and the execution_request process exits. Otherwise, "nack" is replyed and the execution_request continues requesting the execution until it is granted. For example, the action#1 is executed only if the corresponding ack is returned. We have the event "port!Disk!Task!terminated" in the next line of the statement "action#1;". The process_manager issues this event immediately after the request of action#1 is granted. The event forces the successive execution of action#1. This request-response(ack or nack) mechanism controls the execution of the event step by step. Thus we must embed the events and the processes for this request-response mechanism between the lines of the LOTOS code in each object level process.



(a) Non-reflective Language



(b) Reflective Language

Figure 8: Structural Feature of Specification by Reflective Languages

Figure 8 shows the difference between RLOTOS code and LOTOS one schematically.

In the MINIX example, we have modeled separately a process controller named process_manager (a process in layer 1) and the processes (in layer2, 3, and 4) controlled by the controller. If we use a non-reflective language, we should embed into the the controlled processes the mechanism to control themselves by the information obtained from the controller. See the statements printed in sans serif fonts in LOTOS code, and the shaded parts in Figure 8. It results in the complex descriptions which are difficult for us to understand. If we use a reflective language, this control mechanism is hidden behind the reflective procedures and we can concentrate on specifying the essential behaviour of the controlled processes. The description of the object level is simple and we can understand easily what processes are performed in MINIX.

The essential point of the reflective specification is the explicit separation of the meta properties such as control characteristics from the object level properties of the system. Thus we have not employed the communications between the object level processes and the meta processes in our examples. The reflective languages have much expressive power. However the complicated communications between the object level processes and the meta processes bring the specification writers and the readers into confusion. The restriction that communications between object level

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[8], 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 separative 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

[1] *Problem Set for the 4th International Workshop on Software Specification and Design: Proc. of 4th International Workshop on Software Specification and Design*, 1987.

[2] ISO 8807. *Information processing systems — Open Systems Interconnection – LOTOS — A formal description technique based on the temporal ordering of obsevational behaviour*, 1989.

[3] ISO/IEC JTC1/SC21/WG1 N1180. *Contribution on Enhancements to LOTOS*, 1992.

[4] N.G. Leveson and J.L. Stolzy. Safety Analysis Using Petri Nets. *IEEE Trans. on Software Engineering*, 13(3):386–397, 1987.

[5] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In *Proc. of ACM OOPSLA '92*, 1992.

[6] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[7] K. Ohmaki, K. Takahashi, and K. Futatsugi. A LOTOS simulator in OBJ. In J. Quemada, J. Manas, and E. Vazquez, editors, *Formal Description Techniques III*, pages 535–538, 1991.

[8] M. Saeki, T. Kaneko, and M. Sakamoto. A Method for Software Process Modeling and Description using LOTOS. In *Proc. of 1st International Conference on the Software Process*, pages 90–104, 1991.

[9] S. Shlaer and S.J. Mellor. An Object-Oriented Approach to Domain Analysis. *ACM SIGSOFT Software Engineering Notes*, 14(5):66–77, 1989.

[10] B.C Smith. Reflection and semantics in Lisp. In *Proc. of 12th ACM Sympo. on POPL*, pages 23–35, 1984.

[11] H. Sugano. Concurrent Logic Language and Reflective Computation in Distributed Environments. In *Proc. of International Workshop on New Models for Software Architecture '92*, pages 69–74, 1992.

[12] J. Tanaka. An Experimental Reflective Programming System written in GHC. *Journal of Information Processing*, 14(1), 1991.

[13] A.S. Tanenbaum. *Operating Systems — Design and Implementation.* Prentice Hall, 1987.

[14] C.A. Vissers, G. Scollo, and M. Sinderen. Architecture and specification style in formal descriptions of distributed systems. In *Protocol Specification, Testing and Verification VIII*, pages 189–204, 1988.

[15] T. Watanabe and A. Yonezawa. Reflective Computation in Object-Oriented Concurrent System and Its Applications. In *Proc. of Fifth IWSSD*, pages 56–58, 1989.