Research Report ISIS-RR-96-9E

Reflective Extension of LOTOS and Its Applications

Takanori Ugai† and Motoshi Saeki‡

June 1996

†International Institute for Advanced Study
of Social Information Science,
FUJITSU LABORATORIES LTD.
1-9-3, Nakase 1-Chome, Mihama-ku,Chiba-shi,Chiba 261, Japan
e-mail: ugai@iias.flab.fujitsu.co.jp

‡Dept. of Electrical and Electronic Engineering,
Tokyo Institute of Technology,
Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
e-mail: saeki@cs.titech.ac.jp

Reflective Extension of LOTOS and Its Applications

Takanori Ugai† Motoshi Saeki‡

†International Institute for Advanced Study
of Social Information Science,
FUJITSU LABORATORIES LTD.
1-9-3, Nakase 1-Chome, Mihama-ku,Chiba-shi,Chiba 261, Japan
e-mail: ugai@iias.flab.fujitsu.co.jp

‡Dept. of Electrical and Electronic Engineering,
Tokyo Institute of Technology,
2-12-1 Ookayama Meguro-ku, Tokyo 152, Japan
e-mail: saeki@cs.titech.ac.jp

Abstract

This paper presents a method for handling exceptional behaviour expressions in a system using RLOTOS (Reflective LOTOS), which is an extension of LOTOS (Language Of Temporal Ordering Specification). We embedded *reflection* or *reflective computation* facilities into the behaviour expressions of LOTOS and produced an RLOTOS interpreter system in the C language, which includes LOTOS executor support. The *Reflective System* has the concept of a *meta level* and an *object level*. The meta level system deals with object-level description as data and executes the object-level description. Users can control the meta-level system by describing a special process, called the *reflective process*. We also show a method of handling exceptional behaviour expressions in RLOTOS to control the Alternating Bit Protocol. We describe normal behaviour expressions at the object level and exceptional behaviour expressions at the meta level. This example will help us to clarify the usefulness of separating descriptions into the meta level and the object level.

Keywords : LOTOS, Reflection, Alternating Bit Protocol, process algebra

1 Introduction

LOTOS (Language Of Temporal Ordering Specification), as developed for the formal specification of communication systems[1]. It has powerful constructs for describing concurrency, non-determinism, synchronous and asynchronous interaction, and interruption. LOTOS has been standardized by the International Standard Organization, and many practical application examples have been reported. Enhancements to the LOTOS language such as object oriented extensions have also been studied [2].

Reflection or *reflective* computation [3, 4, 5] in a program is a mechanism to access and modify the execution states of its executor has. If a computational system has reflective capabilities, it becomes possible to catch the current state while executing the program and to take the appropriate action according to the obtained information. This type of language has several large scale applications, for example operating systems. In general, reflective facilities prescribe two-level descriptions of programs — object level descriptions and meta level descriptions. Meta level descriptions control the execution of the object level description. The object level descriptions, which are "programs" in the usual sense, are treated as data by the meta level descriptions. In the reflective language, the meta level description is also described in the same language as the object level language. Thus we should have an execution control mechanism, such as the description of an interpreter, which can be defined in the same framework as the object level language.

The specifications written in LOTOS consist of behaviour specifications, which define the observable interaction sequences of a system, and data specifications. We have introduced reflection facilities to the behaviour specification part. Our new language derived from LOTOS is called *Reflective LOTOS* or *RLOTOS* [6]. We specify the control mechanism, i.e., the LOTOS interpreter, by using LOTOS constructs explicitly. For example, we can change the order of action in a behaviour expression at the object level, or determine and set the variable values and control the action at the object level.

In out implementation the object-level behaviour consistently connects with the data in meta level behaviour. The meta-level descriptions in RLOTOS for the object level are specifications in standard LOTOS, and handled as data. We explore a method to represent LOTOS descriptions as terms of ACT ONE, which is the data description language of LOTOS. We also discuss a method to control the interpreter of a meta process, and define "reflective gates" and a "reflective process." The "reflective gates" pass the computational information of the object-level process such as the current state, the events occurring at the next time, and the next possible states.

LOTOS is an executable language, and RLOTOS is also executable. We have implemented an interpreter system for RLOTOS. This interpreter system is based on LOTOS interpreter written in LOTOS.

In LOTOS it is difficult to separate exceptional behaviour and normal behaviour. In RLOTOS, we can describe the exceptional system behaviours at the meta level and normal behaviours at the object level. Using the monitoring capabilities of the RLOTOS hierarchiesed system, we can describe a debug system at the meta level. We can also implement a rapid prototype system for our enhanced LOTOS. We can easily create a prototype executor for an enhanced version of LOTOS, e.g. CELOTOS[7] because the operational semantics of the languages can be described at the meta level. The descriptions of meta processes also provides a facility for verification and validation of object-level code.

In this paper, we mainly discuss methods to separate th exceptional behaviour from normal

behaviour using the meta- and object-level description. We also describe the Alternate Bit Protocol[8] in RLOTOS. In Section 2, we show a method to describe the Alternate Bit Protocol in LOTOS. In Section 3, we introduce our new language RLOTOS derived from LOTOS. In Section 4, we cover language semantics and execution, followed by four application examples to validate our approach. In Finally, in Section 5, we show a method for handling exceptional behaviour expressions in the Alternate Bit Protocol using RLOTOS. This example will help us to clarify the usefulness of separating behaviour into the meta- and object-level descriptions.

2 LOTOS

2.1 Overview of LOTOS

Descriptions written in LOTOS consist of behaviour specifications and data specifications. The formal semantics of LOTOS are based on Communication and Concurrency Specifications (CCS) [9] for behaviour specifications and on the algebra of Abstract Data Types (ADT) [10] for data specifications.

The behaviour 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 hierarchically broken-down into several *subprocesses*. The atomic unit of interaction is an *event*. The *process description* contains the *behaviour expression* defining the observable behaviour of the process.

LOTOS has several constructors for behaviour expressions as shown in Table 1. For example, the expression "$((a; b; exit)[](c; b; exit))|||((a; b; exit)[](c; d; exit))$" is a behaviour expression.

A labeled transition system is used to model the behaviour of behaviour expressions [1]. It provides the formal interpretation for LOTOS behaviour expressions. For example, the behaviour expression "a;B" can execute the action "a" and then turn into expression "B". Thus transition rules can be defined by a *current* behaviour expression, an event which will occur next, and a *next* behaviour expression. In the above example, the transition rule can be expressed as "$a; B \xrightarrow{a} B$". For example, the behaviour expression "$((a; b; exit)[](c; b; exit))|||((a; b; exit)[](c; d; exit))$" can be interpreted as a sequences of the actions, "a, b, a, b" , "a, a, b, b" , "c, b, a, b" , "c, a, b, b" , "c, b, c, d" , "c, c, b, d" , or "c, c, d, b".

2.2 Alternating Bit Protocol

In this section we present an example of a LOTOS specification, namely a communication protocol. This protocol is often referred to as the Alternating Bit Protocol (AB Protocol) in literature. A communication protocol concerns the transmission of data through an unreliable channel, such that – despite the unreliability – no information will get lost. The operating environment for such a system is shown in Figure 1.

In Figure 1, the *Sender* is the sender passes data elements *msg* to the *Receiver* through the unreliable *Channel_K*. After having received the data element, the *Receiver* will acknowledge *Sender* through *Channel_L*, which is also unreliable. In practice we usually find that *Channel_K* and *Channel_L* are physically the same medium, as shown in Figure 1. We show the description in LOTOS to define this protocol such that no information will get lost. That is to say the behaviour of the entire process – apart from the communications at the internal ports *port_1*, *port_2*, *port_3*, and *port_4* – satisfies the description in LOTOS.

Table 1: Operators of behaviour expressions.

Constructor	Function
a;B	Sequential execution a to B
B1 $\rangle\rangle$ B2	Sequential execution B1 to B2
B1 [ ] B2	Selection of B1 or B2
B1 || B2	Synchronize B1 and B2
B1 ||| B2	Interleave B1 and B2
B1 $\|[g_1, \ldots, g_n]\|$ B2	Synchronize on gates
B1 [$\rangle$ B2	Disable B1 by B2
hide $g_1, \ldots, g_n$ in B	Hide gates
stop	Inaction
exit	Terminate

a: Action
B, B1, B2: Behaviour expression
$g_1, \ldots g_n$: Gates

```
specification one_element_buffer[in_port,out_port]:noexit
type    message is
sorts   msg
opns    m0              : -> msg
        m               : msg -> msg
(* m0 , m(m0), m(m(m0)) .... are messages *)
endtype
behaviour
        buffer[in_port,out_port]
where
process buffer[in_port, out_port]: noexit :=
        in_port ? x:msg; out_port !x; buffer[in_port,out_port]
endproc
endspec
```

This process behaves externally as a one-element buffer.

Appendix A gives the detailed solution to this problem [11].

In the general description, both normal behaviour and exceptional behaviour are described in a flat structure. Therefore it is difficult to understand the total normal behaviour. In this description, it is easy to debug of the parts (*Sender*, *Channel*, and *Receiver*), but it is difficult to debug the total behaviour. To easily understand the system behaviour it is better to describe the normal behaviour and the exceptional behaviour separately. In the next section, we provide a new language in which the user can describe the exceptional behaviour separately.

3 RLOTOS

We have designed RLOTOS (Reflective LOTOS) with the reflective mechanism to allow the system to access and modify the state of the executor. In RLOTOS, we can describe a reflective process for the meta level to control the object-level descriptions. Furthermore, we produced an RLOTOS interpreter system in the C language and LOTOS executor support.

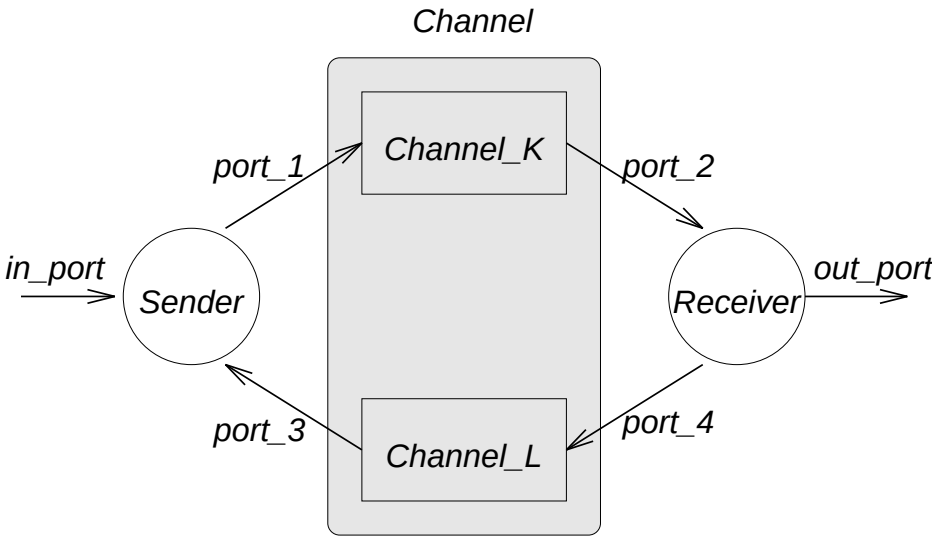


Figure 1: Alternating Bit Protocol.

3.1 Reflective computation in LOTOS

3.1.1 Object level and meta level relationship

In general, reflective facilities cause the introduction of two-level descriptions of programs — object level descriptions and meta level descriptions. The object level has both a notational and execution relationship to the meta level. We defined the notational relationship between the object-level components and their meta notations in LOTOS, as shown in Table 2. All the identifiers in the object level are the same as in the meta level. Assume that an object level process "buffer" has a gate "input". At the meta level, "buffer" is the term of type `ProcessID` and "input" is the term of type `NameId`.

For the executional relationship between the object level and the meta level, the object level descriptions should be consistently related to the execution of the meta level. Consider when an event occurs at the gate "input" at the object level, and an event occurs at gate "outg" with a value "$input$" at the meta level. In a LOTOS interpreter (see Section 4.1 for details), the events at the object level correspond to the terms of ACT ONE that are passed through the special gate "outg" by the interpreter at the meta level. For example, a sequential occurrence of events a, b and c at the object level correspond to a sequential occurrence of events $outg!a$, $outg!b$ and $outg!c$ at the meta level, as shown in Figure 2. That is to say, our system preserves the temporal order of events between the object level and meta level. The behaviour expression in the object level is handled as an ADT module of the type *Exp*. Each operation in the behaviour expression is represent in Table 3 (see Section 4.2 for details).

3.1.2 Reflective process and reflective gates

In this section we will detail the control mechanism for the object level of descriptions from the meta level. The meta level has a special process, called the meta process, which controls the execution at the object level. There can only be one meta process per level. A *meta process* has a *reflective process* and *reflective gates*. Figure 3 shows the internal structure of the meta

Table 2: Relationship between object level and meta level.

Object level	Meta level (data type)
Gate a	Term (Name) a
Process p	Term (PName) p
Variable x	Term (Var) x
Occurrence of Event a	outg !a

Table 3: Operators of behaviour expressions.

Constructor	ACT ONE representation
a;B	pre(*a*, B^*)
B1 $\rangle\rangle$ B2	ena($B1^*$, $B2^*$)
B1 [ ] B2	cho($B1^*$, $B2^*$)
B1 || B2	sync($B1^*$, $B2^*$)
B1 ||| B2	para($B1^*$, $B2^*$)
B1 $\|[g_1, \ldots, g_n]\|$ B2	gen($B1^*$, *gate-set*, $B2^*$)
B1 [$\rangle$ B2	dis($B1^*$, $B2^*$)
hide $g_1, \ldots, g_n$ in B	hid(*gate-set*, B^*)
stop	act_stop
exit	act_exit

a: action
B, B1, B2: behaviour expression
$g_1, \ldots . g_n$: gates
gate-set: ACT ONE representation of a set of gates
a: ACT ONE representation of the action for a
B^*,$B1^*$, $B2^*$: ACT ONE representation of behaviour expression

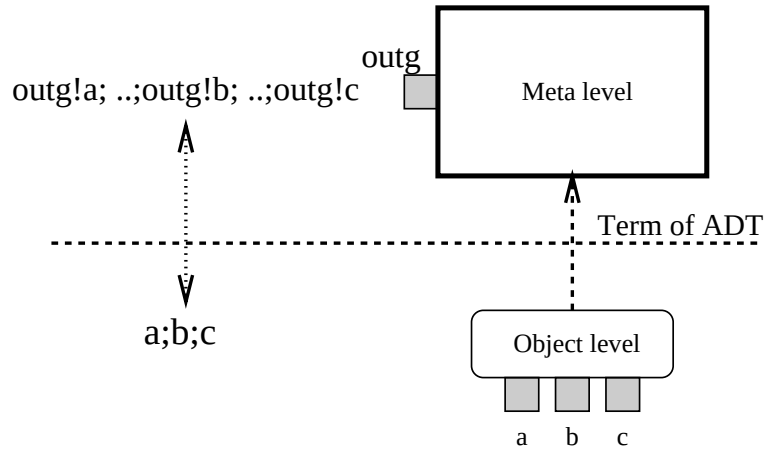


Figure 2: Event sequence in object level and meta level.

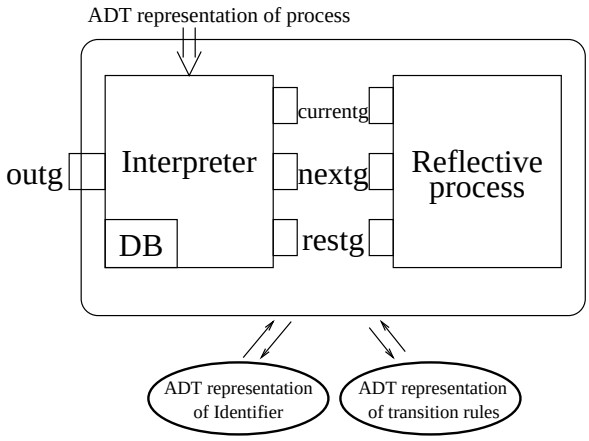


Figure 3: Internal structure of a meta process

process. The meta process contains an interpreter process which simulates a labeled transition system of LOTOS. The interpreter process has three special gates called *reflective gates* which are used for referring to information about execution and for changing its behaviour. A behaviour expression to be currently executed is passed through the reflective gate "currentg" to the *reflective process*. The reflective gates "nextg" and "restg" are used for the next event and the behaviour expression after the next event. If you change the execution of the object level description, you send to the reflective gates a new event or a new behaviour expression, which may be different from the one specified by the labeled transition rules of LOTOS. The interpreter process at the meta level executes the object-level process based on information passed though *reflective gates*. We can control the object-level process with a *reflective process*. An example of the code of a meta process written in LOTOS is as follows.

```
process meta-process[outg](exp) :=
  hide currentg,nextg,restg in
  interpreter[currentg,nextg,restg,outg](exp)
 |[currentg,nextg,restg]|
  reflective_process[currentg,nextg,restg]

  where
    process interpreter[currentg,nextg,restg,outg](exp):=
      choice x:ActSet []
         [IsIn(x,head(exp))] -> currentg !exp; nextg !x; restg !rest(x,exp);
                                nextg ?x:ActSet; outg !x;  restg ?z:Exp;
                                interpreter[currentg,nextg,restg,outg](z)
         []
         [NotIn(x,head(exp))] -> interpreter[currentg,nextg,restg,outg](exp)
  endproc
endproc
```

A *reflective* process in a meta process computes the next event and the next behaviour expression from the current behaviour expression which is received through the "currentg" gate. This computation may be performed independently of the labeled transition rules of LOTOS. The reflective process sends the computation results to the interpreter process through the "nextg" and "restg" gates. As for state transitions, the computation of the reflective process

takes precedence over that of the interpreter. That is to say, the reflective process controls the execution of the corresponding object-level descriptions. If you do not need to change the execution of an object-level description, you define a reflective process to send the received next event and next behaviour expression back to the interpreter without modification. If you do not define a meta process explicitly, it is employed as a default reflective process.

```
/* Reflective process for a trivial default */
process reflective_process[currentg,nextg,restg] :=
  currentg ?x : Exp ; nextg ?y : Act ;  restg ?z : Exp ;
  nextg !y ;  restg !z ; reflective_process[currentg,nextg,restg]
end proc
```

As mentioned earlier, reflective computations in LOTOS control its interpreter with reflective gates and a reflective process. In our reflective mechanism, we can consider the meta level of the reflective processes. This level controls the behaviour of the reflective processes in the same way. It allows us to establish a meta level for the meta level, and to construct the infinite reflective tower according to the object-meta hierarchy.

3.2 Organization of RLOTOS

In our reflective mechanism, we can consider that in RLOTOS we describe the object level behaviour expressions, the reflective process of the meta level, and the reflective processes of the meta-meta level (or meta-meta-meta, ... etc. level) as shown in Figure 4. Reflective processes will use “currentg”, “nextg”, and “restg” to communicate with the interpreter process of each level, and get or put internal information of the interpreter process. The syntax of RLOTOS is based on LOTOS. Figure 5 shows the structure of RLOTOS descriptions. In Fig. 5, the “process meta-level” stands for the reflective process of meta level and “process meta-meta-level” stands for the reflective process of meta meta level. The object level behaviour expression must be described but the reflective processes may not be specified because the default reflective process is given. In the “reflective process for meta level” in Fig. 5, we specify the RLOTOS code to control object level behaviour. For example, the operation to change the order of action in the object-level behaviour expression and to get or set the value of variables and actions can be specified.

4 Implementation of RLOTOS interpreter

We have implemented the RLOTOS interpreter with the SEDOS LOTOS interpreter[12]. An interpreter process is defined by standard LOTOS, as shown in Section 4.1, and the converter from object-level descriptions to term representations is implemented in the C language. The RLOTOS interpreter transforms RLOTOS descriptions into LOTOS ones, embedding the interpreter process. The transformed descriptions can be executed by the SEDOS LOTOS interpreter when the converter translates object-level descriptions to the term representations and inputs them.

4.1 Basic idea of LOTOS interpreter in LOTOS

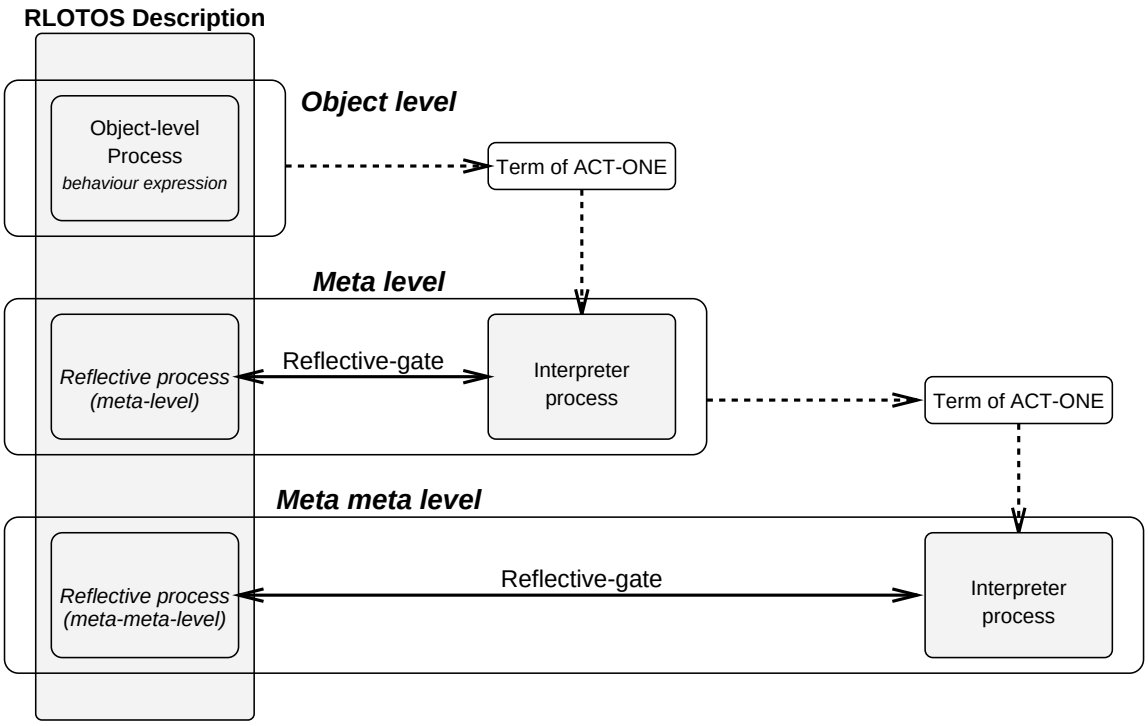


Figure 4: RLOTOS description and hierarchy of levels.

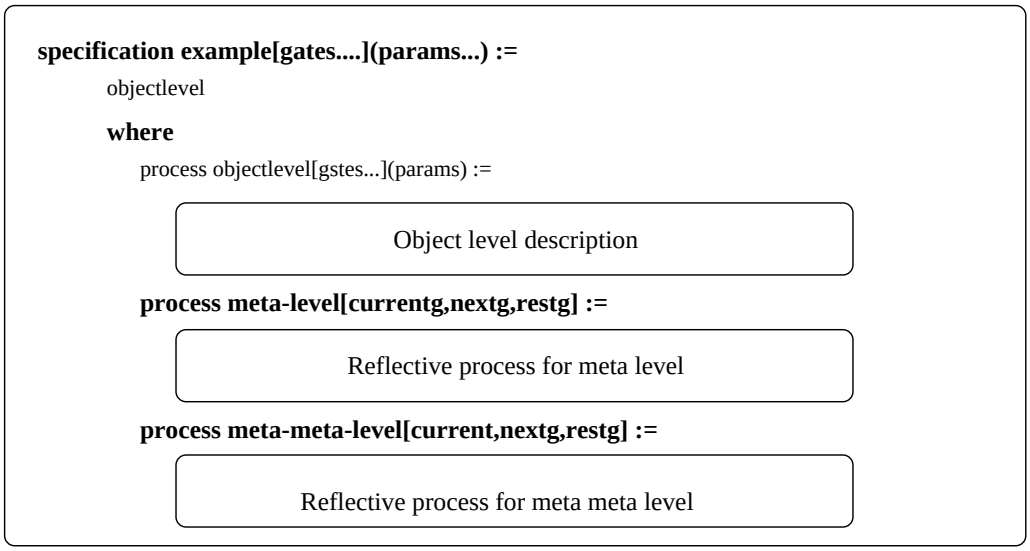


Figure 5: RLOTOS description.

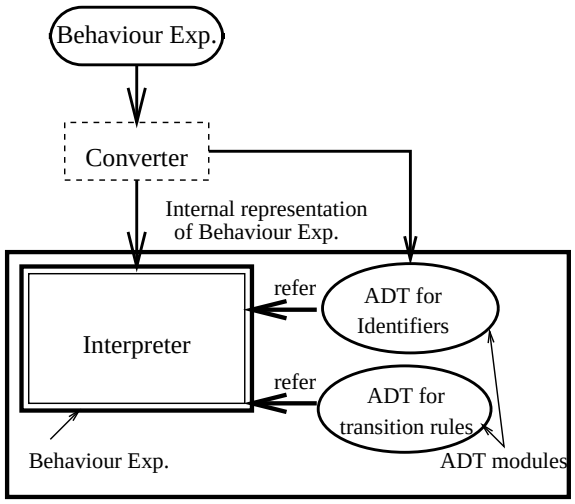


Figure 6: Organization of a LOTOS Interpreter

We can define a LOTOS interpreter for the term-represented LOTOS descriptions. Figure 6 shows the organization of the LOTOS interpreter written in LOTOS. The *converter* in the figure produces the term representation of the LOTOS description and ADT modules for the various kinds of the identifiers appearing in the description. The term representation produced is passed to the *interpreter* described with LOTOS, and then executed. The *interpreter* can be defined as a LOTOS process using *action prefix*, *choice*, and *general choice* operators. In this interpreter, the non-determinism of LOTOS is defined by the non-determinism characteristics of the *general choice* operator. The interpreter is defined as

```
process Interpreter[outg](exp) :=
   [eq(head(exp),{})] -> stop
   []
   [ne(head(exp),{})] -> (choice x: ActSet []
     ([IsIn(x,forall(head(exp)))] -> (outg! x;Interpreter[outg](rest(x,exp)))
      []
      [NotIn(x,forall(head(exp)))] -> Interpreter[outg](exp)))
end proc
```

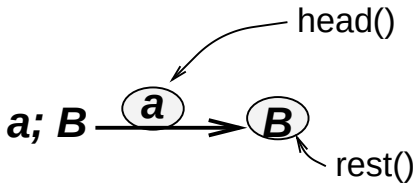
The “interpreter” process sequentially puts out values for the names of the action denotations which occur on executing the behaviour “exp” expression through the “outg” gate.

4.2 Term representation of behaviour expressions

In reflective languages, the description for the meta level is also described by the language itself. The object level source code or descriptions are handled as data in the meta level. Since *terms* in ACT ONE express data in LOTOS, we established a method to define LOTOS descriptions with the *terms*.

The behaviour expressions can be defined as terms of an abstract data type `Exp` as shown in Table 3[13].

For example, the behaviour expression “$((a;b;exit)[](c;b;exit))|||((a;b;exit)[](c;d;exit))$” can be represented by the term, where a,b,c,d are the representations of the actions :



head(pre(a', B')) = {a'};
rest(a', pre(a', B')) = B';

a': ACT ONE representaion of the action a
B': ACT ONE representaion of the behaviour expression B

Figure 7: Labelled transition system for head() and rest() functions

$$para(cho(pre(a, pre(b, act\_exit)), pre(c, pre(b, act\_exit))),\\ cho(pre(a, pre(b, act\_exit)), pre(c, pre(d, act\_exit))))$$

We define the term representation of an action as

A !*value* ?*variable*:*type*...

$\Downarrow$

atr(A, oup(*value*)+inp(*variable*, *type*)+...)

atr: Name, SymbolList, Symbol $\rightarrow$ NameId
oup: Symbol $\rightarrow$ Symbol
inp: ValueId, SortId $\rightarrow$ Symbol

NameId : type of event in LOTOS
SortId : type of sortname of variables

The operator "atr" is used for constructing a term representing an action denotation from a gate identifier, identifier for input, and value expression for output. Oup and inp are functions for representing value expression and identifier in term form respectively. For example, the action $a!x$ is represented by the term "atr(a, oup(x))". We use a as the abbreviation of "atr(a,...)" whenever possible.

4.3 Representation of operational semantics

A labeled transition system is used to model the behaviour of behaviour expressions[1]. It provides the formal interpretation for LOTOS behaviour expressions. For example, the behaviour expression "a;B" can execute the action "a" and then turn into expression "B." Thus transition rules can be defined by a *current* behaviour expression, an event which will occur next, and a *next* behaviour expression. In the above example, the transition rule can be expressed as "$a; B \xrightarrow{a} B$". A LOTOS interpreter should compute a set of events which can occur next, and a *next* behaviour expression which is a result of a transition. We define two functions "$head$" and "$rest$" on the ADT "Exp." Figure 7 shows how the function "$head$" computes from a behaviour expression, the events which can occur next, and how "$rest$" outputs the *next* behaviour expression from the *current* behaviour expression and event. That is to say, the labeled transition rule "$B_1 \xrightarrow{a} B_2$" iff $\{a'\} \in head(B_1')$ and $rest(\{a'\}, B_1') = B_2'$ where B_1' and

B_2' are term representations of B_1 and B_2 respectively, and a' is a pair of the action denotation a and its occurrence position in B_1.

The "$head$" and "$rest$" functions handle the terms of type "Act," that consists of terms of type "NameId" that show the event name, and the term that shows the id-number. The "$head$" function returns a set of terms of type "ActSet" that shows a set of events that occur in synchronously. The inputs of "$rest$" are a term of type "ActSet",which is a term element that the "$head$" function returns, and a term of type "Exp," which is the behaviour expression. The functions "$head$" and "$rest$" can be defined in the ADT module "Exp" according to the labeled transition rules of LOTOS.

```
type Exp is ...
sorts Exp
opns
  pre    : NameId,Exp -> Exp
  para   : Exp,Exp -> Exp
  cho    : Exp,Exp -> Exp
  (* constructors for behaviour expression *)
  head   : Exp,Number -> ActSetSet
  rest   : ActSet,Exp -> Exp
eqns
forall E1,E2:Exp, A:Act, I:NameId, S1,S2:ActSet
ofsorts ActSetSet
  head(pre(I,E1)) = InsertSet(Insert(I,{}),{}) ;
          (* rule for action prefix *)
  head(cho(E1,E2)) = Union(head(E1),head(E2)) ;
          (* rule for choice operator *)
  head(para(E1,E2)) =  Union(head(E1),head(E2)) ;
          (* rule for interleaving operator *)
  ....
ofsorts Exp
 rest(InsertSet(A,{}),pre(I,E1)) = E1 ;
        (* rule for action prefix *)
 IsIn(S1,head(E1)) => rest(S1,cho(B1,E2)) = rest(S1,E1);
 IsIn(S1,head(E2)) => rest(S1,cho(E1,E2)) = rest(S1,E2);
       (* rule for choice operator *)
 IsIn(S1,head(E1)) => rest(S1,para(E1,E2))=para(rest(S1,E1),E2);
 IsIn(S1,head(E2)) => rest(S1,para(E1,E2))=para(E1,rest(S1,E2));
       (* rule for interleaving operator *)
  ....
endtype
```

For example,

head(para(cho(pre(a,pre(b,act_exit)),pre(c,pre(b,act_exit))),
cho(pre(a,pre(b,act_exit)),pre(c,pre(d,act_exit))))) =
{a,a,c,c};
rest(a, para(cho(pre(a,pre(b,act_exit)),pre(c,pre(b,act_exit))),
cho(pre(a,pre(b,act_exit)),pre(c,pre(d,act_exit))))) =
para(pre(b,act_exit),cho(pre(a,pre(b,act_exit)), pre(c,pre(d,act_exit))));

5 RLOTOS application of Alternating Bit Protocol

We considered and described the following four applications:

- Separating normal behaviour and exceptional behaviour
- Describing a debug system at the meta level
- Implementing of a rapid prototype system for enhanced LOTOS s (e.g. CELOTOS [7])
- Programming a meta process to verify and validate object-level code

In this paper we will present the alternating bit protocol in RLOTOS. Figure 8 shows the state diagram for the Alternating Bit Protocol.

In Fig. 8, the "+" mark stands for incorrect data (*undef* or *e_msg*).

In the description of the Alternating Bit Protocol in Section 2.2, it is difficult to understand which parts are normal behaviour and which parts are exceptional behaviour. It is difficult to understand the behaviour of the system when the system works normally.

In RLOTOS, we describe only the normal behaviour in the object level, and in the meta level we describe the exceptional behaviour. The normal operation of the system is in Fig. 9 whereas exceptional operation is shown in Figure 10.

The description for the normal behaviour of the system is given as

```
(* Alternating Bit Protocol *)
specification alternating_bit_protocol[in_port,out_port]:noexit
(* data type modele is same as the LOTOS description. *)
behaviour
    hide port_1, port_2, port_3, port_4 in
        sender[in_port, port_1, port_3](0) |[port_1, port_3]|
            channels[port_1, port_2, port_3, port_4] |[port_2, port_4]|
            receiver[out_port, port_2, port_4](1)
where
process sender[in_port, port_1, port_3](bit:parity): noexit :=
   in_port ? x:msg; port_1 ! dat(x,bit); port_3 ? y:parity;
        sender[in_port, port_1, port_3](rev(bit))
endproc
process receiver[out_port, port_2, port_4](bit:parity): noexit :=
   port_2 ? x:msg_bit ; out_port ! getmsg(x); port_4 ! rev(bit);
        receiver[out_port, port_2, port_4](rev(bit))
endproc
process channels[port_1, port_2, port_3, port_4]: noexit :=
        channel_K[port_1, port_2] ||| channel_L[port_3, port_4]
where
    process channel_K[port_1, port_2]: noexit :=
        port_1 ? x:msg_bit ; i;port_2! x;channel_K[port_1, port_2]
    endproc
    process channel_L[port_3, port_4]: noexit :=
        port_4 ? x:parity ; i;port_3 ! x; channel_L[port_3, port_4]
    endproc
endproc
endspec
```

This description is very simple and so we can easily understand the behaviour of the system. Now we describe the exceptional behaviour at the meta level. In Fig. 10 an error occurs at *port_2*.

```
process meta-level[currentg, nextg, restg] : noexit :=
    exception[currentg, nextg, restg](m0)
```

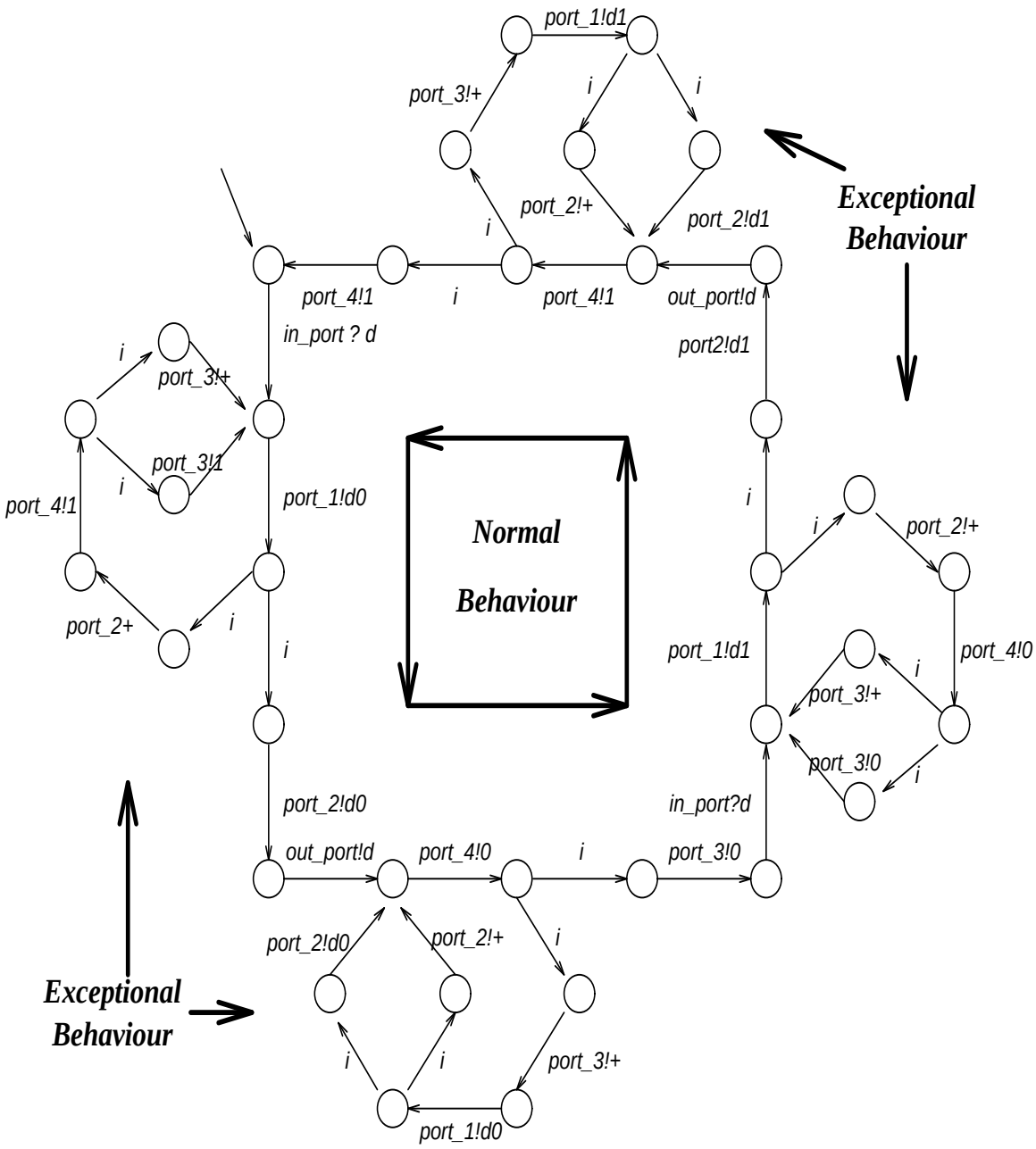


Figure 8: System state diagram for Alternating Bit Protocol

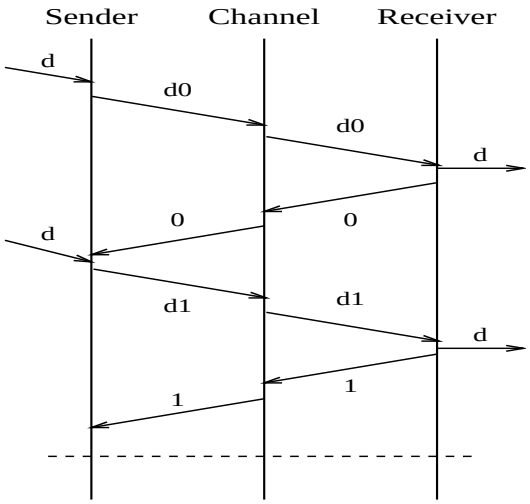


Figure 9: Normal operating behaviour of Alternating Bit Protocol

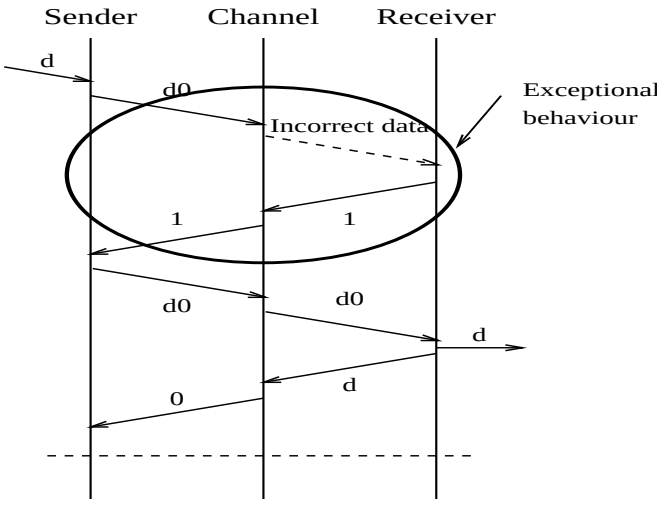


Figure 10: Exceptional operating behaviour of Alternating Bit Protocol

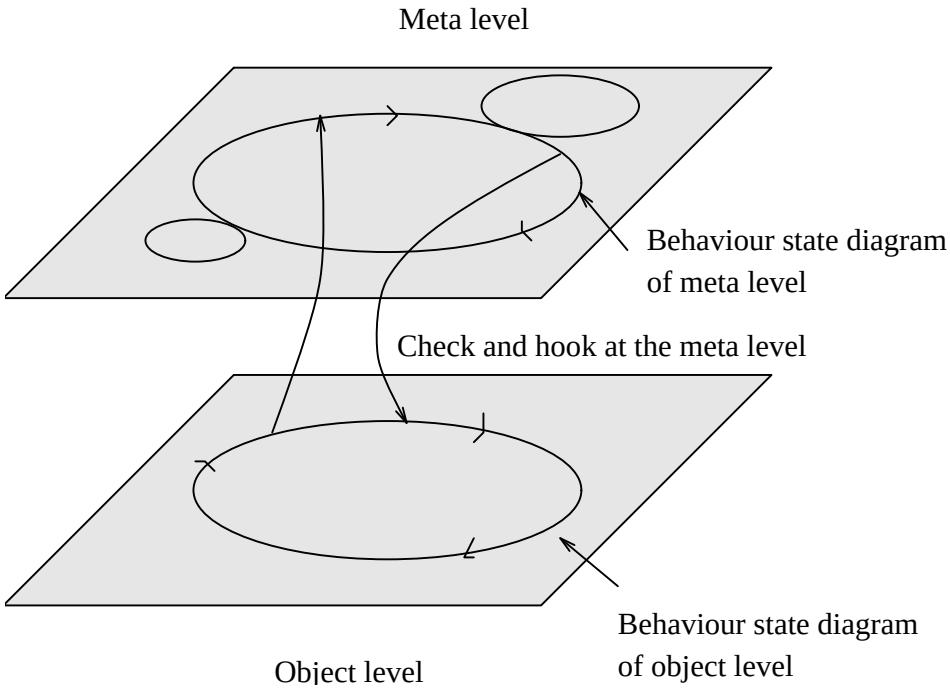


Figure 11: Object level and meta level

```
where
process exception[currentg, nextg, restg] : noexit :=
    currentg ?x:Exp; nextg ?y:Act restg ?z:Exp;
    ([getname(y) = port_2] ->
        (nextg!y; restg !x; exceptional[currentg, nextg, restg](m)
         [] nextg ! atr(port_2,outp(undef)) ;
             restg ! pre(atr(port_4,outp(rev(getnum(y)))),
                      pre(atr(port_3,outp(rev(getnum(y))))),
                       pre(atr(port_1,outp(getdata(y)))),x);
             exceptional[currentg,nextg,restg]
           (* the datum is incorrect in port_2 *)
                     :
           (* When the datum is incorrect in port_1, port_3 , port_4
                the system works almost same as when incorrect at  port_2 *)
endproc
endproc
```

In the code above, *getname(x)* is an ADT operation to get the name of the event passed through “next” on “EXP.”

In *port_2*, a datum is incorrect. The meta level then hooks the behaviour of the object level as shown in Figure 11. The meta level resends the datum and return to the normal behaviour. The system requests the re-send of the datum to the sender when the datum is incorrect in any of the ports.

In this example, we have not discussed a *time out*, however we can easily extend the Alternating Bit Protocol to use *time out*.

6 Futurework

In RLOTOS, the user can easily describe the object level simply, however it is difficult to describe the meta level process. We will provide the method of the description of the meta level process. Our reflective facilities are embedded not in the ADT part but in the behaviour expression part. Embedding the reflection of ADTs and the "Unit" of reflection, i.e. process, are for further study.

7 Conclusion

In this paper we have introduced and discussed the reflective LOTOS language. We have defined a reflective mechanism in LOTOS with "reflective gates" and "reflective processes." The reflective process for the meta-level process can control object-level processes by communicating with the interpreter process through reflective gates. We have also shown the usefulness of RLOTOS by describing the alternating bit protocol and several other examples.

Acknowledgements

The authors have benefited from discussions with Prof. C. A. Vissers and Mr. L. F. Pires of University of Twente. We especially thank to them for their careful reading the earlier version of this paper. We also thank to Mr. Takeshi Hiroi of the Tokyo Institute of Technology for discussions and suggestions.

References

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

[2] Korean Experts. An Object Oriented Extensition of LOTOS for Distributed Processing. Technical report, SG-ODP K116, 1989.

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

[4] Takuo Watanabe and Akinori Yonezawa. Reflective Computation in Object-Oriented Concurrent System and Its Applications. In *Proc. of Fifth IWSSD*, pp. 56–58, 1989.

[5] Jiro Tanaka. An Experimental Reflective Programming System written in GHC. *Journal of Information Processing*, Vol. 14, No. 1,, 1991.

[6] Motoshi Saeki, Takeshi Hiroi, and Takanori Ugai. Reflective Specification : Applying A Reflective Language To Formal Specification. In *Proc. of Seventh IWSSD*, pp. 204–213. IEEE Computer Society, Dec 1993.

[7] Wilfried H. P. van Hulzen and Paul A. J. Tilanus. LOTOS Extended with Clocks. In S. T. Vuong, editor, *Formal Description Techniques II*, pp. 179–191. Elsevier Science Publishers B.V. (North-Holland), 1990.

[8] J.A. Bergstra and J.W. Klop. Verification of an alternating bit protocol by means of process algebra. In W.Bibel and K.P.Jantke, editors, *Math. Methods of Spec. and Synthesis of Software System '85*, pp. 9–22. Springer Verlag, 1985. LNCS 215.

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

[10] H. Ehrig and B. Mahr. *Foundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[11] J.C.M. Baeten and W.P.Weijland. *PROCESS ALGEBRA*. CAMBRIDGE UNIVERSITY PRESS, 1990.

[12] P. H. van Eijk. The Design of a Simulator Tool. In *The Formal Description Technique LOTOS*. North-Holland, 1989.

[13] Kazuhito Ohmaki, Koichi Takahashi, and Kokichi Futatsugi. A LOTOS simulator in OBJ. In E. Vazquez J. Quemada, J. Manas, editor, *Formal Description Techniques III*, pp. 535–538. Elsevier Science Publishers B.V. (North-Holland), 1991.

[14] C. A. Vissers. FDTs for Open Distributed Systems, a Retrospective and a Prospective View. In *Protocol Specification, Testing and Verification X*, pp. 341–362. North-Holland, June 1990.

Appendix A. ABProtocol in LOTOS

Sender will read a datum **d** at *in_port* and passes on a sequence **d0, d0, d0, ...** of copies of this datum, appended with a bit **0**, to *Channel_K* until an acknowledgement **0** is received at *port_3*. Then, the next datum is read, and sent on together with a bit **1**, likewise the acknowledgement is the reception of a **1**. The following data element has a **0** in turn, and so forms the **0,1** alternating bit pattern.

Channel_K is the data transmission channel, passing on frames of the form **d0,d1**. *Channel_K* may corrupt data, however, passing on *e_msg* (an error message) we assume that the incorrect transmission of **d** can be recognized, for instance, by using a checksum.

Receiver receives frames **d0,d1** from *Channel_K*, sending on **d** to *out_port* (if this was not done earlier), and the acknowledgement **0** (resp. **1**) is sent to *Channel_L*. *Channel_L* is the acknowledgement transmission channel, and passes bits **0** (resp. **1**), received from *Receiver*, on to *Sender*. Channel_L is also unreliable, and may send on **undef** instead of **0** (resp. **1**). Now we present the processes *Sender*, *Channel_K*,*Channel_L*,*Receiver* by means of recursive specifications.

```
(* Alternating Bit Protocol *)
specification alternating_bit_protocol[in_port,out_port]:noexit
type    message is
      (* a module for messages *)
endtype
type    parity  is
sorts   parity
opns    0, 1, undef     : -> parity
        rev             : parity -> parity
eqns    ofsort parity
        rev(0)  = 1 ;
        rev(1)  = 0 ;
endtype
type    mes_bit is message, parity
sorts    msg_bit
opns    e_msg           : -> msg_bit
        dat             : msg, parity -> msg_bit
        getmsg          : msg_bit -> msg
        get_p           : msg_bit -> parity
eqns    forall d:msg, a:parity
        ofsort msg
        getmsg(dat(d, a))  = d;
        ofsort parity
        get_p(dat(d, a))   = a;
endtype
behaviour  (* sender, channel and receiver are synchronized *)
    hide port_1, port_2, port_3, port_4 in
        sender[in_port, port_1, port_3](0) |[port_1, port_3]|
            channels[port_1, port_2, port_3, port_4] |[port_2, port_4]|
            receiver[out_port, port_2, port_4](1)
where
process sender[in_port, port_1, port_3](bit:parity): noexit :=
        in_port ? x:msg; send1[in_port, port_1, port_3](x,bit)
where
  process send1[in_port, port_1, port_3](d:msg,bit:parity): noexit :=
    port_1 ! dat(d,bit) ; port_3 ? ans : parity;
    ([ans = bit] -> sender[in_port, port_1, port_3](rev(bit))
      (* normal behaviour *)
    [][ans= undef] -> send1[in_port, port_1, port_3](d,bit)
      (* illegal behaviour *)
    [][ans= rev(bit)] -> send1[in_port, port_1, port_3](d,bit))
      (* illegal behaviour *)
  endproc
endproc
process receiver[out_port, port_2, port_4](bit:parity): noexit :=
   port_2 ? x:msg_bit ;
```

```
   ([get_p(x)= rev(bit)] -> out_port ! getmsg(x);
                            port_4 ! rev(bit);
                            receiver[out_port, port_2, port_4](rev(bit))
     (* normal behaviour *)
    [][x= e_msg] -> port_4 ! bit ; receiver[out_port, port_2, port_4](bit)
     (* exceptional behaviour *)
    [][get_p(x) = bit] -> port_4 ! bit ;
                          receiver[out_port, port_2, port_4](bit))
     (* exceptional behaviour *)
endproc
(* channels process consists on two channels *)
process channels[port_1, port_2, port_3, port_4]: noexit :=
        channel_K[port_1, port_2] ||| channel_L[port_3, port_4]
where
    process channel_K[port_1, port_2]: noexit :=
        port_1 ? x:msg_bit ; i;
     ((port_2 ! e_msg; exit)  (* exceptional behaviour *)
     [] (port_2! x; exit)) (* normal behaviour *)
        >> channel_K[port_1, port_2]
    endproc
    process channel_L[port_3, port_4]: noexit :=
        port_4 ? x:parity ; i;
     ((port_3 ! undef; exit) (* exceptional behaviour *)
     [] (port_3 ! x; exit)) (* normal behaviour *)
        >> channel_L[port_3, port_4]
    endproc
endproc
endspec
```

In this description, *dat()* is the operation which produces the data with *msg* and *parity* and *getmsg()* gets the *msg* in a datum.