A LOTOS Specification of a "Transit-Node" *

Laurent Mounier
INRIA projet SPECTRE – VERIMAG†
Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE
e-mail: `Laurent.Mounier@imag.fr`

Abstract

This report describes the formal specification and verification of a "Transit-Node", an abstraction of a routing component of a communication network. First, an informal definition of the Transit-Node, initially proposed within the RACE project SPECS, is formally described using the LOTOS language. Then, it is verified following a model-based approach: the LOTOS specification is translated into a finite LTS, and its correctness is checked on this model. Practically, all the verifications has been performed using CÆSAR-ALDÉBARAN, a toolbox for the verification of LOTOS programs.

This work was carried out in the framework of the French project VTT (Verification, Types and Time), those goal was to compare and evaluate different verification methods on a same case study.

Introduction

Formal verification is a part of system design those purpose is to prove statically, on a system *description*, some of the properties expected on its run-time behaviour. Therefore, it offers an attractive way to increase the confidence often required on many system implementations which are both hard to design, and impossible to test exhaustively. Thus, a considerable need for formal verification methods appeared in many areas of computer science, such as asynchronous circuit, communication protocols and distributed software design.

Formal verification has been largely studied on a theoretical point of view, and several approaches have been proposed, with their associated description formalisms, and sometimes leading to automatic verification methods. However, each approach is specific, and therefore is usually well suited for the specification and verification of particular aspects of the system description.

*This work was supported by the French VTT Project (Verification, Types and Time)

†VERIMAG is a joint laboratory of CNRS (Centre National de la Recherche Scientifique), INPG (Institut National Polytechnique de Grenoble), UJF (Université Joseph Fourier, Grenoble-I) and VERILOG SA. VERIMAG is associated with IMAG (Institut d'Informatique et de Mathématiques Appliquées de Grenoble). SPECTRE is a project of INRIA (Institut National de Recherche en Informatique et Automatique).

This work was carried out within a French project called VTT (for Verification, Types and Time), which was intended to compare and evaluate several formal verification methods. More precisely, the goal of this project was to specify and verify a same case study through different approaches, in order to better identify choices and difficulties which are proper to a given approach, from the ones common to each of them.

The case study chosen for this purpose is a description of a routing component of a communication network, called a "Transit-Node". This example was initially defined in the RACE project 2039 SPECS (Specification Environment for Communication Software). It consists of a simple transit node where messages arrive, are routed, and leave the node.

This informal specification describes a Transit-Node at a rather high abstraction level. In fact, it can be viewed as a list of requirements indicating *what* a correct Transit-Node should be, but it does not provide any informations on *how* to design it. Therefore, design choices have to be made during the formal specification phase, which are influenced by the description formalism used.

One can object that this case study is very simple on a complexity point of view (number of parallel components, size of each components, ... etc.). In fact, it is true that it does not reflects all the difficulties and problems arising when trying to verify a real-world system. Nevertheless, it remains interesting for two main reasons:

- most of the properties to verify are representative of the ones usually required on more realistic examples ;
- the Transit-Node is supposed to function within a given *environment*, not explicited in the informal description, but which must however be taken into account during the verification phase.

This report is organised as follows: first, we recalled in section 1 the principles of the verification method we choose to formally verify this case study, and we briefly present the toolset used in practice. In section 2 we give the informal description of the Transit-Node, as initially defined in the SPECS project, and in section 3 we propose a formal specification in LOTOS. The verification phase is then described in the subsequent sections: section 4 discusses the Transit-Node environment, and section 5 to 8 are devoted to the correctness proof of the LOTOS specification, with respect to each requirements appearing in the informal description.

1 The verification framework

We present in this section the framework we used to specify and verify the Transit-Node. More precisely, we first briefly recall the principles of the so-called "model-based" verification method, and then we describe the software environment chosen in practice, namely the CÆSAR-ALDÉBARAN toolbox.

1.1 The model-based approach

A possible approach for parallel program verification consists in translating a program description into a suitable abstract model, which represents its exhaustive behaviour. Then, the verification phase can be carried out on this abstract model, deciding whether it satisfies or not the program *specification* (a set of properties defining its expected behaviour). Several formalisms can be used to express the program to be verified, provided that they have a well-defined operational semantics. Thus, a possible abstract model is the labeled transition system (LTS, in the sequel) associated to each program by this semantics. Finally, note that whenever this LTS is finite, automatic verification can be considered.

Two classes of model-based verification methods are usually distinguished in practice, depending on the formalism used to express the specification:

logical specifications:
: They are expressed in terms of formulas of a *temporal logic*. Such logics allow to characterise overall properties on the program behaviour, such as mutual exclusion, liveness, deadlock freeness, ..., etc. In this case a *satisfaction relation* is defined between the program model and the logical formulas, and any decision procedure for this relation offers a verification method.

behavioural specifications:
: They describe the *expected behaviour* of the program observed from a certain *abstraction level*, for instance when considering as *visible* only a subset of its actions. Thus, such properties can be straightly expressed by a LTS, or in any description language which can be compiled into a LTS (for instance the one used to describe the program itself). As both the program and its specifications can be represented by LTS, the verification phase consists in comparing these two LTS using suitable equivalence or preorder relations: the program satisfies its specification whenever the two LTS are related. Therefore, any decision procedure for such relations offers a verification method.

Usually these two approaches are considered as complementaries: some properties are easier to express within a logical formalism, whereas some others can be straightly represented by a LTS. However, for practical reasons (mainly tool availability), this verification of the Transit-Node was performed only in terms of behavioural specifications.

Within this last approach, several relations have been proposed to compare the LTS representing the program with the more abstract one representing its specification. Among these relations, an important class relies on the *bisimulation relation* family [Par81], which offers a notion of equality between two program behaviours defined on the same set of visible actions. More precisely, from a general definition of bisimulation, several relations can be built, differing on how internal actions (i.e., non visible ones) are dealt with during the LTS comparison. Two of them will be used throughout this report, and this choice rely on the distinction usually made between *safety* and *liveness* properties [Lam77]:

safety equivalence [BFG$^+$91]:
: This relation (noted $\approx_s$) exactly characterises the *safety properties* of systems: two LTS are related if and only if they verify the same set of safety properties. Note that a specification can be considered as a safety property when it defines a *superset* of the expected behaviour of the program: in other words, the program can be considered as correct whenever its behaviour remains *included* in the one defined by its specification.

branching bisimulation [GW89]:
: This relation (noted $\approx_b$) is stronger than safety equivalence (i.e., it distinguishes more LTS).

Moreover, whenever the two LTS under comparison are *livelock-free* (i.e., they do not contain any circuit of internal actions), then this relation preserves *liveness properties* [NV90]: if these two LTS are related, then they satisfy the same set of liveness properties. Note that a specification should be considered as a liveness property whenever the behaviour it defines must be *eventually* executed by the program to ensure its correction.

Finally, note that (several) efficient decision procedures exist for both relations.

1.2 The Cæsar-Aldébaran toolbox

To carry out the verification from a practical point of view, the software environment we used is the Cæsar-Aldébaran toolbox [FGM+92], which implements a model-based verification method for the Lotos language[1].

The toolbox is organised around two main components, a Lotos compiler, Cæsar and a LTS equivalence checker, Aldébaran.

Cæsar:

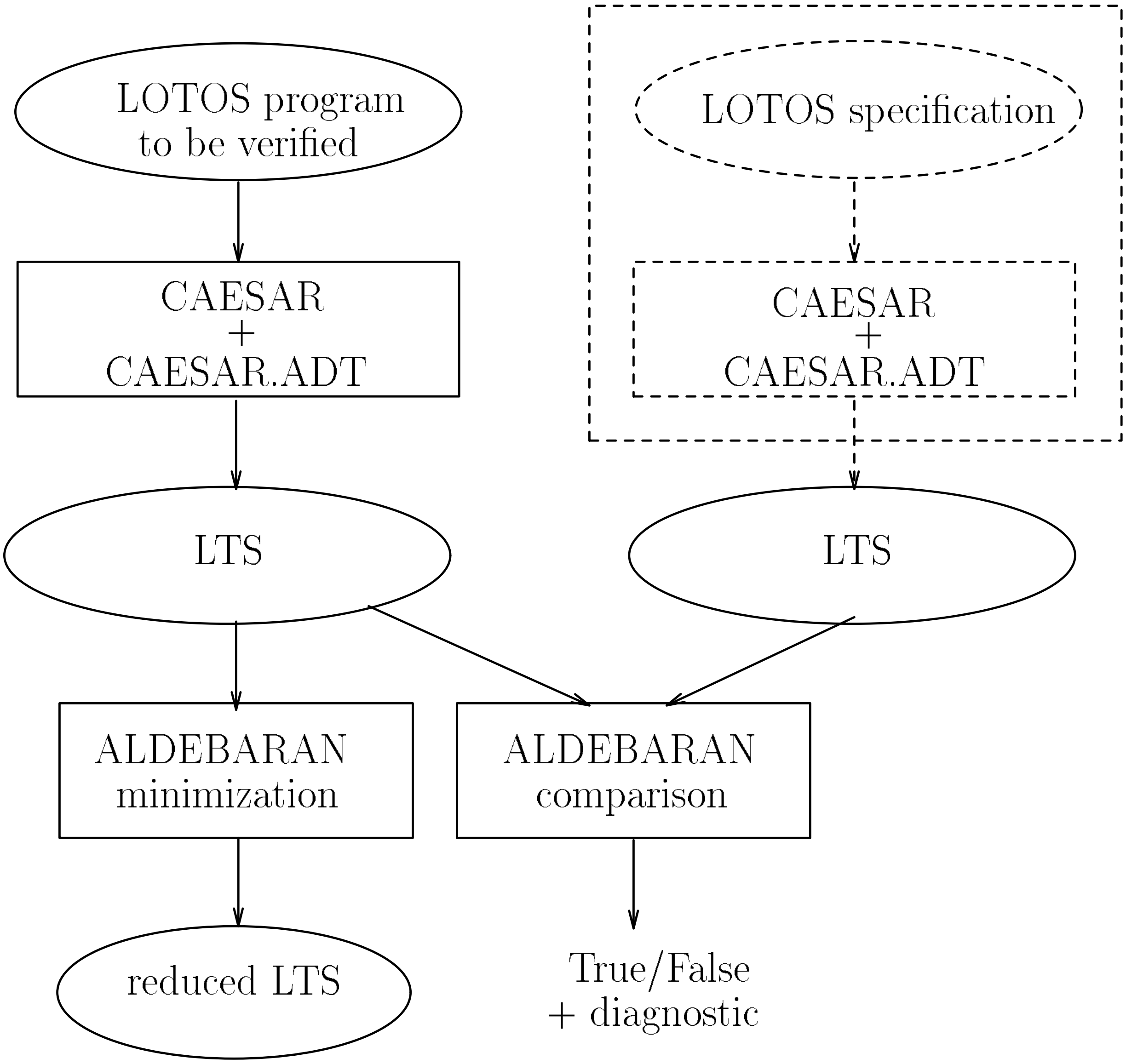
The Cæsar tool allows to compile a Lotos description into a finite LTS. As the Lotos language is build upon *process algebra* such as CCS [Mil80] and CSP [Hoa78], its operational semantics can be defined in terms of LTS. Therefore, a straightforward compilation technic could be considered. However, an original feature of Cæsar is to use an extended petri net as an intermediate form during the compilation process. This leads to a more efficient tool, even for large Lotos programs [GS90]. Finally, data parts of Lotos programs, expressed within an abstract data type formalism (the Act-One language), is handled by an independent tool, Cæsar.Adt, which produced a C code library used by Cæsar.

Aldébaran:

The Aldébaran tool allows to reduce or compare LTS with respect to various equivalence or preorder relations such as strong bisimulation, safety equivalence or preorder, and branching bisimulation. Furthermore, whenever the LTS under comparison are not related, a diagnostic is computed in terms of erroneous execution sequences. Finally, deadlock and livelock detection can also be performed. Note that two algorithms are implemented within the tool to deal with Lotos programs for those the underlying LTS is too large to be straightly generated [FM91, FKM93]. Such algorithms are usually required when considering realistic examples.

The architecture of the toolbox is summarised on the figure below:

[1] Lotos is a standard formal description technique for protocol and distributed systems [ISO87].



Architecture of the CÆSAR-ALDÉBARAN toolbox

2 The "Transit-Node" case study

We give here the informal specification of the Transit-Node, as it appeared in the SPECS project [2]. We also indicate each modifications and corrections brought to this initial description during the VTT project.

clause 1 The system to be specified consists of a transit node with:

- one *Control Port-In*

[2]its structuring in clauses and their numbering is due to A. Arnold, and it facilitates further references.

- one *Control Port-Out*
- N *Data Ports-In*
- N *Data Ports-Out*
- M *Routes Through*

(The limits of N and M are not specified.)

clause 2 **(a)** Each port is serialized.
(b) All ports are concurrent to all others. The ports should be specified as separate, concurrent entities.
(c) Messages arrive from the environment only when a *Port-In* is able to treat them.

clause 3 The node is "fair". All messages are equally likely to be treated, when a selection must be made,

clause 4 and all messages will eventually transit the node, or be placed in the collection of faulty messages.
*** Modification July 93 *** and all data messages will eventually transit the node, or become faulty. ***

clause 5 *Initial State :* one *Control Port-In,* one *Control Port-Out.*

clause 6 The *Control Port-In* accepts and treats the following three messages:

(a) *Add-Data-Port-In-&-Out(n)*

> gives the node knowledge of a new *Port-In(n)* and a new *Port-Out(n)*. The nodes commences to accept and treat messages sent to the *Port-In*, as indicated below on *Data Port-In.*

(b) *Add-Route(m),(n(i),n(j)...))*

> gives the node knowledge of a route associating route m with *Data-Port-Out(n(i),n(j),...).*

(c) *Send-Faults*

> routes all saved faulty messages, if any, to *Control Port-Out.* The order in which the faulty messages are transmitted is not specified.
> *** Modification July 93 *** routes some messages in the faulty collection, if any, ...***

clause 7 A *Data Port-In* accepts and treats only messages of the type :

- *Route(m).Data*

  (a) The *Port-In* routes the message, unchanged, to any one (non determinate) of the *Data Ports-Out* associated with route m.
  *** Modification march 93 *** ... to any one (non determinate) of the open *Data Ports-Out* associated with route m. If no such port exists, the message is put in the faulty collection.***
  (b) (Note that a *Data Port-Out* is serialized – the message has to be buffered until the *Data Port-Out* can process it).
  (c) The message becomes a faulty message if its transit time through the node (from initial receipt by a *Data Port-In* to transmision by a *Data Port-Out*) is greater than a constant time T.

clause 8 *Data Ports-Out* and *Control Port-Out* accept messages of any type and will transmit the message out of the node. Messages may leave the node in any order.

clause 9 All faulty messages are saved until a *Send-Faults* command message causes them to be routed to *Control Port-Out*.
*** Modification July 93 *** All faulty messages are eventually placed in the faulty collection where they stay until ...***

clause 10 Faulty messages are **(a)** messages on the *Control Port-In* that are not one of the three commands listed, **(b)** messages on a *Data Port-In* that indicate an unknown route, or **(c)** messages whose transit time through the node is greater than T.

clause 11 [3] **(a)** Messages that exceed the transit time of T become faulty as soon as the time T is exceeded.
(b) It is permissible for a faulty message not to be routed to *Control Port-Out*
*** Modification July 93 *** ... not to be routed to *Control Port-Out* by a *Send-Faults* command***
(because, for example, it has just become faulty, but has not yet been placed in a faulty message collection),
(c) but all faulty messages must eventually be sent to *Control Port-Out* with a succession of *Send-Faults* commands.

clause 12 It may be assumed that a source of time (time-of-day or a signal each time interval) is available in the environment and need not be modeled within the specification.

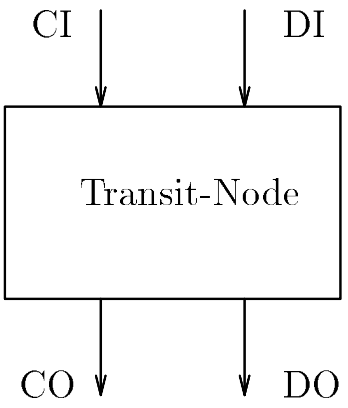
[3] The stucturation in three parts of this clause dates back to july 93

3 A Lotos specification of the "Transit-Node"

This section is devoted to the formal specification of the Transit-Node using the Lotos language. We first present the general architecture of the specification, and then we specify each of its components, following the informal description recalled in section 2.

3.1 Architecture of the specification

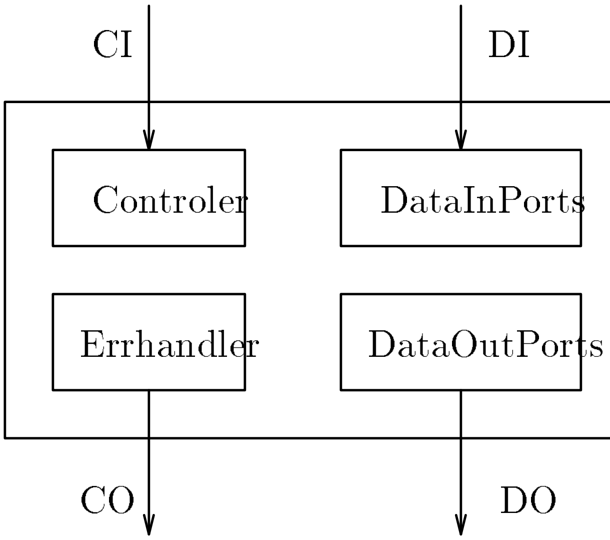
At the higher level (i.e., from an external point of view), the Transit-Node can be viewed as a "black box", interacting with its environment either by its control ports, or by its data ports. Thus, if we call `CI` (*resp.* `CO`) the interacting point associated to the *Control Port-In* (*resp.* to the *Control Port-Out*) and `DI` (*resp.* `DO`) the one associated to the *Data Port-In* set (*resp.* to the *Data Port-Out.* set), we obtain the following representation:



External representation of the Transit-Node

According to the informal description, all ports are concurrent to all others, and they should be specified as separate entities (clause 2 (a)). Consequently, in the Lotos specification, the ports will be modeled by four distinct processes:

- process `Controller`, for *Control Port-In*
- process `ErrHandler`, for *Control Port-Out*
- process `DataInPorts`, for the set *Data Port-In*
- process `DataOutPorts`, for the set *Data Port-Out.*



Process decomposition of the LOTOS specification

Before discussing how these processes communicate together, we first specify each of them in turn. Then, we come back to the whole specification in section 3.3.

3.2 Modeling the ports

As usual in LOTOS, data and control parts of processes are specified separately.

3.2.1 Data types

Associated to the ports, several data sets have to be specified (e.g., the set of existing routes, the collection of faulty messages, ..., etc.). We give here an overview of the abstract data types used in the specification to represent these sets.

Data messages, port and route identifiers:

Since the data part of the messages is not relevant, data messages are represented by natural numbers called "envelopes", and belonging to a sort `Env`. Similarly, data ports and routes are identified by natural numbers, belonging respectively to sorts called `PortNo` and `RouteNo`.

Message lists:

Lists of data messages are represented by a sort `EnvList`, build from two constructor operators,

- `emptyl`, representing the empty list,
- `insert(l,e)`, representing the list `l` extended by the envelope `e` of sort `Env`,

and four non-constructor operators,

- `head(l)` and `tail(l)`, delivering respectively the head and the tail of list `l`,
- `e IsIn l`, indicating whether envelope `e` belongs or not to list `l`,

- `remove(e,l)`, which returns a new list equal to `l` but without envelope `e` (which may or not belong to `l`).

Port sets:

Sets of data port identifiers are represented by a sort `PortSet`, build from two constructor operators,

- `emptyset`, representing the empty port set,
- `add(p,ps)`, representing the port set `ps` extended by the new port `p` of sort `PortNo`,

and two non-constructor operators,

- `p IsIn ps`, returning whether port `p` is or not present in port set `ps`,
- `ps1 includes ps2`, returning whether port set `ps2` is included or not in port set `ps1`.

4. Route sets:

Sets of route identifiers are represented by a sort `RouteSet`, build from two constructor operators,

- `emptyrl`, representing the empty route set,
- `insert(r, ps, rs)`, representing the route set `rs` extended by the new route `r` (of sort `RouteNo`) which defines the port set `ps` (of sort `PortSet`). In other words, `ps` is the port set associated to the new route `r`,

and with three non-constructor operators,

- `r IsIn rs`, indicating whether route `r` is or not present in the route set `rs`,
- `route(r,rs)`, returning the port set associated to route number `r` in the route set `rs` (assuming route `r` belongs to `rs`),
- `update(r,ps,rs)`, returning a new route set equal to `rs`, but where the port set associated to route `r` is replaced by the port set `ps` (assuming route `r` belongs to `rs`).

The formal specification of these data types can be found in appendix.

3.2.2 The Control Port-In

This port accepts and treats the control messages received by the Transit-Node, but it also has to manage two sets `rl` and `ps`, representing respectively the routes and data ports defined so far. More precisely, the set `ps` ensures that no data port is created twice (clause 6 (a)), and the set `rl` is intended to keep track of the current status of each route already defined (this set will be consulted by other processes).

On reception of a control message, the behaviour of this process is the following:

Add-Data-Port-In-&-Out(p):
: if p does not already belong to `ps` then a notification is sent to processes `DataInPorts` and `DataOutPorts` (by a *Create-Port(p)* internal message), and the new port p is added to the port set `ps`. Otherwise, no action is performed.

Add-Route(r,s):
: either the new route r is inserted to the set `rl`, or this set is simply updated if this route was already defined.

Send-Faults:
: a notification is sent to the process `ErrHandler` (by an *Error-Out* internal message), which manages the buffer of faulty messages

Other-Command: (representing any incorrect control message)
: a notification is sent to the process `ErrHandler` (by a *Control-Error-In* internal message), indicating that a new incorrect control message has been received.

As we shall see later, the process `Controller` may also receive requests from the other processes regarding the status of a given route (*Route-Query (r)* internal message). If this route has been correctly defined, then its associated port set s is returned (*Route-Answer(r,s)* internal message), otherwise the *Route-Answer(r, ∅)* internal message is returned.

The LOTOS specification of process `Controller` is then the following:

```
process Controller [CI, erri, erro, crep, rq, ra]
                                (rl:RouteSet, ps:PortSet) : noexit :=

(* Valid commands from control-port-in *)
  CI !Add_Data_Port ?n:PortNo [not (n IsIn ps)] ;
        crep !n ; (* port creation *)
            Controller [CI, erri, erro, crep, rq, ra] (rl, add(n,ps))
[]
  CI !Add_Route ?r:RouteNo ?s:PortSet ;
        AdRoute [CI, erri, erro, crep, rq, ra] (rl, ps, r, s)
[]
  CI !Send_Faults ;
        erro ; (* error out *)
           Controller [CI, erri, erro, crep, rq, ra] (rl, ps)

(* Other command from control-port-in *)
[]
  CI !Other_Command ;
        erri ; (* error in *)
           Controller [CI, erri, erro, crep, rq, ra] (rl, ps)

(* Route query from other processes *)
[]
  rq ?r:RouteNo  (* route query *)
   [(r IsIn rl) and not(route(r, rl) == emptyset)
    and (ps includes route(r, rl))] ;
        ra !route(r, rl) !r !e ;  (* route answer *)
           Controller [CI, erri, erro, crep, rq, ra] (rl, ps)
```

```
[]
  rq ?r:RouteNo  (* route query *)
   [not(r IsIn rl) or route(r, rl) == emptyset
    or not(ps includes route(r, rl))] ;
         ra !Unknown_Route !r !e ; (* route answer *)
             Controller [CI, erri, erro, crep, rq, ra] (rl, ps)
endproc
```

where the subprocess AdRoute is defined by:

```
process  AdRoute [CI, erri, erro, crep, rq, ra]
                 (rl:RouteSet, ps:PortSet, r:RouteNo, s:PortSet) : noexit :=

  [r IsIn rl] -> (* update an existing route *)
        Controller [CI, erri, erro, crep, rq, ra] (update(r, s, rl), ps)
[]
  [not (r IsIn rl)] -> (* add a new route *)
        Controller [CI, erri, erro, crep, rq, ra] (insert(r, s, rl), ps)
endproc
```

3.2.3 The Data Port-In

The process DataInPorts manages the *Data Port-In* set. Here again, to satisfy clause 2 (a), all data ports-in are modelised as independent processes, without any interaction between each others.

After its activation (notified by the Controller process), each data port-in process (called Inport in the sequel) becomes able to receive and treat data messages. More precisely, on reception of a *Route(r).Data* message, the behaviour of an Inport process is the following:

1. an internal *Route-Query(r)* message is sent to the process Controller, to determine the current status of route r,
2. - if the route exists and is well defined (reception of a *Route-Answer(r,s)* internal message from the Controller), then the data message is delivered to *one* (non deterministically chosen) data port-out referenced by s. This is performed by sending an *Input-Output* internal message to the DataPortOut process.
   - Otherwise, the message is considered faulty (clause 7 and 9), and it is transmitted to the ErrHandler process (by sending a *Data-Error-In(e)* internal message)

In the LOTOS specification, we left aside the dynamic creation of the Inport processes. In fact, if such a solution would have been possible in theory, it is not allowed by the CÆSAR compiler since it may lead to an infinite state model. Furthermore, the maximal number of data ports which could be created is supposed to be bounded by a value N (according to clause 1). Consequently, all Inport processes are created statically (one per potential data port-in), and they are activated on reception of a corresponding *Create-Port* internal message:

```
process  DataInPorts [DI, crep, rq, ra, erri, io] : noexit :=
```

```
   (crep !0 of PortNo ; Inport [DI, rq, ra, erri, io] (0 of PortNo))
                              |||
   (crep !1 of PortNo ; Inport [DI, rq, ra, erri, io] (1 of PortNo))
                              |||
                              ...
                              |||
   (crep !N of PortNo ; Inport [DI, rq, ra, erri, io] (N of PortNo))
endproc
```

The specification of an Inport process is the following:

```
process Inport [DI, rq, ra, erri, io] (n:PortNo) : noexit :=

DI !n ?e:Env ?r:RouteNo ;
   rq !r ; (* route query for the Controller process *)
      (
         ra ?s:PortSet !r ; (* positive answer *)
             choice outp:PortNo [] [outp IsIn s] -> io !outp !e ;
                Inport [DI, rq, ra, erri, io] (n)
       []
         ra ?er:ErrorCode !r ; erri !e ; (* negative answer *)
             InPort [DI, rq, ra, erri, io] (n)
      )
endproc
```

3.2.4 The Data Port-Out

The process DataOutPorts manages the *Data Port-Out* set. As for the data ports-in, each port is modeled by an independent process, without any interaction with the other data ports-out. After its activation, each of these processes (called OutPort in the sequel) must be able to perform the following tasks:

- to store in an initially empty buffer l the data messages transmitted by the process DataInPorts (i.e., on reception of an *Input-Output* internal message)
- to route these messages outside the Transit-Node (and then to remove them from buffer l)
- to manage the time spent in the Transit-Node by each data message, and to declare faulty the ones for which this value is greater than a constant time Tc. Each timed-out message must then be sent to the ErrHandler process (using a *Timeout* internal message).

As for Inport process the set of OutPort processes are created statically:

```
process  DataOutPorts [DO, crep, io, timeout] : noexit :=

  (crep !0 of PortNo ; OutPort [DO, io, timeout] (0 of PortNo, emptyl))
                              |||
  (crep !1 of PortNo ; OutPort [DO, io, timeout] (1 of PortNo, emptyl))
                              ...
                              |||
  (crep !N of PortNo ; OutPort [DO, io, timeout] (N of PortNo, emptyl))
endproc
```

In its current version, the LOTOS language is not really suitable to specify time considerations on a quantitative point of view. Consequently, we chose to model the "timeout" of a data message by a simple non deterministic choice: any data message with a correct route number can either be routed correctly outside the node or becoming faulty because its transit time within the node exceeded constant Tc. Thus, we preserve the exhaustive behaviour of the Transit-Node, as it is expected from its informal description.

The LOTOS specification of an `OutPort` process is then straightforward:

```
process OutPort [DO, io, timeout] (n:PortNo, l:EnvList) : noexit :=

  io !n ?e:Env [not (e IsIn l)]; (* reception of a data msg from DataInPorts *)
        OutPort [DO, io, timeout] (n, insert(e, l))
[]
  [not (l == emptyl)] ->
    (
       (DO !n !head(l) ; (* emission of a data msg outside the node *)
             OutPort [DO, io, timeout] (n, tail(l)))
    []
       (timeout !head(l) ; (* timed-out data msg ==> faulty *)
             OutPort [DO, io, timeout] (n, tail(l)))
    )
endproc
```

3.2.5 The Control Port-Out

The last port to be modelised is the *Control Port-Out*, represented by the `ErrHandler` process.

The goal of this process is to store the set of faulty messages (control messages or data messages) transmitted by processes `Controller`, `DataInPorts` or `DataOutPorts` through *Control-Error-In*, *Data-Error-In* or *Timeout* internal messages. Moreover, on reception of an *Error-Out* internal message from the `Controller`, all faulty messages have to be sent outside the node on the control port-out.

Here again, to ensure that the model obtained from our LOTOS specification remains finite, we have to ensure that the buffer of faulty messages remains finite itself. Assuming the number of *distinct* messages received by the Transit-Node is finite, a sufficient condition is then to decide that each message (data or control) will be stored only once in this buffer. The solution chosen to implement this buffer is the following:

- faulty data messages are stored in a message list `l`, of sort `EnvList`
- faulty control messages, which are all designed by the unique notation *Other-Command*, are represented by a boolean value `b`: `b` is true if at least one faulty control message is present in the buffer, and is false otherwise.

We can now deduce a LOTOS specification for process `ErrHandler`:

```
process ErrHandler [cerri, derri, timeout, erro, CO]
                                   (b:Bool, l:EnvList) : noexit :=
```

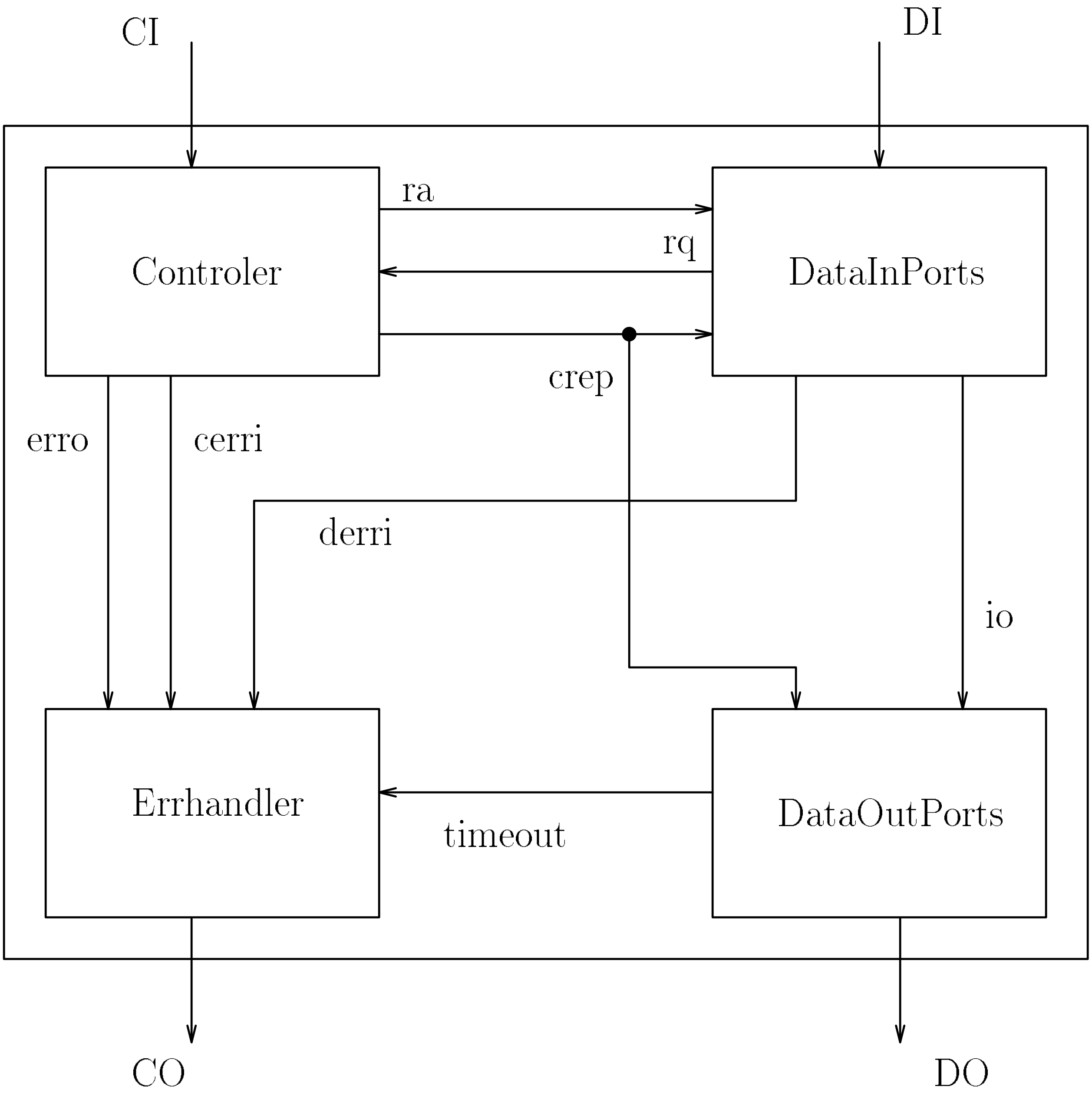
```
  cerri ;
        ErrHandler [cerri, derri, timeout, erro, CO] (true, l)
[]
  derri ?e:Env [not (e IsIn l)] ;
        ErrHandler [cerri, derri, timeout, erro, CO] (b, insert(e,l))
[]
  derri ?e:Env [e IsIn l] ;
        ErrHandler [cerri, derri, timeout, erro, CO] (b, l)
[]
  timeout ?e:Env [not (e IsIn l)] ;
        ErrHandler [cerri, derri, timeout, erro, CO] (b, insert(e,l))
[]
  timeout ?e:Env [e IsIn l] ;
        ErrHandler [cerri, derri, timeout, erro, CO] (b, l)
[]
  erro ;
    CO !b !l ;
        ErrHandler [cerri, derri, timeout, erro, CO] (false, emptyl)
endproc
```

3.3 Communications inside the node

It remains to describe how these processes communicate inside the Transit-Node. The table below summarises the internal communication messages which have been introduced so far, together with their corresponding LOTOS notation.

Designation	LOTOS representation (gate)	Description
Route-Query(r)	`rq !r`	requests for status of route `r`
Route-Answer(r,s)	`ra !s !r`	delivers port set `s` associated to route `r`
Route-Answer(r, $\emptyset$)	`ra !Unknown !r`	indicates that route `r` is not correctly defined
Control-Error-In	`cerri`	puts a control message in the faulty message buffer
Data-Error-In(e)	`derri !e`	puts data message `e` in the faulty message buffer
Error-Out	`erro`	requests for flushing out the faulty message buffer
Create-Port(p)	`crep !p`	creates a new data port `p`
Input-Output(p,e)	`io !p !e`	puts message `e` in the buffer associated to port `p`
Timeout(e)	`timeout !e`	puts timed-out message `e` in the faulty buffer

From the definition of the different processes, it results that communications inside the node are organised as follows:



Internal representation of the Transit-Node

Such an architecture can be straightly expressed in LOTOS:

```
hide  rq, ra, cerri, derri, erro, crep, io, timeout in

(
  (
        Controller [CI, cerri, erro, crep, rq, ra] (emptyrl, emptyset)
                        |[cerri, erro]|
        ErrHandler [cerri, derri, timeout, erro, CO] (false, emptyl)
```

```
  )
              |[crep, rq, ra, derri, timeout]|
  (
       Datainports [DI, crep, rq, ra, derri, io]
                       |[io]|
       Dataoutports [DO, crep, io, timeout]
  )
)
```

4 Model Generation

To ensure the correction of our LOTOS specification of the Transit-Node, it will be necessary to translate it into a finite state model, i.e., a finite LTS. First we show how this initial specification need to be extended to deal with the *environment* of the Transit-Node, and then we discuss the model generation from a practical point of view.

4.1 Modeling the environment

As it appears in its informal description, the Transit-Node represents an *open* system, able to receive any (potentially infinite) number of distinct control or data messages, and in any order. Such a system cannot be represented by a finite LTS. Therefore, we have to limit this too general context, and to turn back to a more restrictive environment.

However, one of the interests of process algebra formalisms (like LOTOS) is to allow a compositional description of such a system (i.e., in a so-called *constraint-oriented* specification style):

1. the Transit-Node is specified as an independent process, without any external constraints (it may not be represented by a finite LTS),
2. the environment is specified as well as an independent process, including all the required constraints,
3. the overall specification is then defined as the parallel composition of these two processes (and therefore can be represented by a finite LTS).

Thus, the description remains modular, and the environment can be easily update without modifying the central part of the specification.

In the remainder of this section we give a LOTOS specification for the environment we chose to consider.

4.1.1 Requirements

The environment has to feed the Transit-Node with control and data messages, which can be considered as tuples. To ensure that the LTS representing the exhaustive behaviour of the system will be finite state, it must satisfy the following requirements:

1. the size of the data domains associated to each message fields must be finite, which is a necessary condition to obtain a finite LTS from a LOTOS program,

2. the number of copies of each data messages (or erroneous control messages) present inside the node must be finite, since these messages have to be stored.

To satisfy requirement 1, it is sufficient to add to the informal node description the extra constraint specifying that the number of distinct data message is finite. Other data domains (presented in section 3.2.1) are already supposed to be finite in the informal description: since the number of ports is supposed to be bounded by N, the numbers of distinct routes and distinct port and route sets are necessarily bounded too.

To show how to satisfy requirements 2, we give the LOTOS specification of two independent processes (the `Data_Msg_Generator` and the `Control_Msg_Generator`), whose respective goals are to deliver data and control messages to the Transit-Node.

4.1.2 The data message generator

To satisfy requirement 2, we modelise the environment in such a way that a data message is sent to the Transit-Node only if no copy of the same message is already present inside it (so the total number of copies inside the node is bound to 1). Consequently, we have to manage a message list `l` to keep track of the data messages already sent to the node and which are still inside. Thus, the process `Data_Msg_Generator` interacts with the node not only through a gate `DI`, but also through gates `DO` and `CO`. Finally, data messages are always sent to the node in any order.

The corresponding LOTOS process is the following:

```
process Data_Msg_Generator [DI, CO, DO] (l:EnvList) : noexit :=

(
  DI ?n:PortNo ?e:Env ?r:RouteNo [not (e IsIn l)] ;
        Data_Generator [DI, DO, CO] (insert(e, l))
        (* send a new data message to the node *)
[]
  DO ?n:PortNo ?e:Env ;
        Data_Generator [DI, DO, CO] (remove(e, l))
        (* message e is no longer in the node => remove it from l *)
[]
  CO ?b:Bool ?l1 ;
        Data_Generator [DI, DO, CO] (intersect(l, l1))
   (* messages of list l1 are no longer in the node => remove them from l *)
)
endproc
```

4.1.3 The control message generator

The main difficulty when specifying the control message generator is to ensure that the resulting environment will be general enough to obtain the exhaustive behaviour of the Transit-Node. Indeed, control messages largely influence this behaviour, and the environment should provide most of the "unexpected" scenarios which may alter the correction of the specification. For instance, a scenario in which a given data port cannot be created must be provided. Unfortunately, it is not possible in practice to modelise a large set of distinct sequences, since it would lead to a too large model.

The solution we adopted is to select for each control message a set of sequences which seems to be relevant, and which leads to a reasonable size model. For a sake of clarity, the LOTOS specification of the `Control_Msg_Generator` process is split into four independent subprocesses, dealing with each control message type. Note that since they are not synchronised, each subprocess can always progress according to its definition.

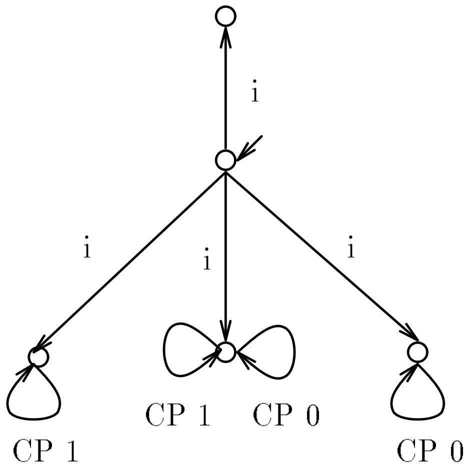
```
process Control_Msg_Generator [CI] : noexit :=

        Add_Port_Generator [CI]  (* data port creation *)
                |||
        Add_Route_Generator [CI] (* route definition *)
                |||
        Send_Faults_Generator [CI] (* Send-Faults requests *)
                |||
        Other_Command_Generator [CI] (* incorrect control messages *)
endproc
```

1. Data port creation:

Regarding the data port creation, we chose to modelise three kinds of sequences, which are non deterministically selected:

- the empty sequence, corresponding to no data port creation,
- for each data port `p`, the infinite sequence creating forever *only* data-port `p`,
- the set of infinite sequences creating forever all the data ports, in any order.

With two data ports, this behaviour corresponds to the following LTS, where `CP 0` and `CP 1` denotes creation of port 0 and 1, and `i` denotes an internal action:



For N data ports, this LTS can be described by the following LOTOS process:

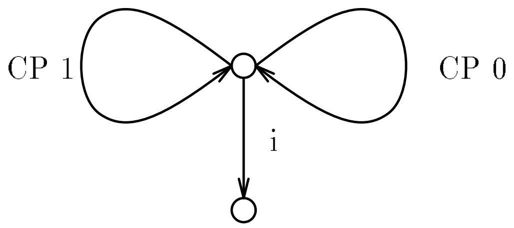
```
process Add_Port_Generator [CI] : noexit :=
```

```
        i ; stop
  []
        i ; Add_One_Port_Generator [CI] (0)
  []
        i ; Add_One_Port_Generator [CI] (1)
  []
        ...
  []
        i ; Add_One_Port_Generator [CI] (N-1)
  []
        i ; Add_All_Port_Generator [CI]
endproc
```

where processes Add_One_Port_Generator and Add_All_Port_Generator are defined by:

```
process Add_One_Port_Generator [CI] (p:PortNo) : noexit :=
  CI !Add_Data_Port !p ;
        Add_One_Port_Generator [CI] (p)
endproc

process Add_All_Port_Generator [CI] : noexit :=

  CI !Add_Data_Port ?p:PortNo ;
        Add_All_Port_Generator [CI]
endproc
```

Other solutions exist to modelise this behaviour, but note that the following (straightforward) LTS is not suitable since it does not provide the scenario in which a given data port creation *cannot* occur:



2. Route definition:

Only one set of infinite sequence is modelised, defining forever all the routes in any order. Note that the scenario for which a given route will never be defined is not specified. On the other hand, any route can be redefined.

```
process Add_Route_Generator [CI] : noexit :=

  CI !Add_Route ?r:RouteNo ?ps:PortSet ;
        Add_Route_Generator [CI]
endproc
```

3. Send-Faults requests:

We modelise both the empty sequence (when no *Send-Faults* message is received by the node) and the infinite sequence for which a *Send-Faults* message is sent forever. These two behaviour are non deterministically chosen:

```
process Send_Faults_Generator [CI] : noexit :=
  i ; stop
[]
  CI !Send_Faults ;
        Send_Faults_Generator [CI]
endproc
```

4. Incorrect control messages:

Any incorrect control message is represented by the *Other-Command* message (hence requirement 2 is satisfied). This message is sent forever to the Transit-Node, which is modelised by the following infinite sequence:

```
process Other_Command_Generator [CI] : noexit :=

  CI !Other_Command
        Other_Command_Generator [CI]
endproc
```

4.1.4 Specification of the whole system

The complete specification of the environment is obtained by interleaving the processes `Control_Msg_Generator` and `Data_Msg_Generator`:

```
process Environment [CI, DI, CO, DO] : noexit :=

        Control_Msg_Generator [CI]
                         |||
        Data_Msg_Generator [DI, CO, DO]
endproc
```

And finally, the specification of the whole system is the full synchronisation of the environment and the Transit-Node specification described in section 3:

```
process Transit_Node_System [CI, DI, CO, DO] : noexit :=

        Transit_Node [CI, DI, CO, DO]
                         ||
        Data_Msg_Generator [CI, DI, CO, DO]
endproc
```

4.2 Practical issues

From this LOTOS specification, a finite LTS can now be generated by the CÆSAR-ALDÉBARAN toolbox. However, it remains to determine the bounds to use in practice for the data domains (maximum number of ports, routes and data messages). Clearly, these values largely influence the size of the resulting LTS.

number of data ports:
We chose to implement a maximum of 2 data ports, labelled from 0 to 1

number of routes:
Once the number of data ports has been established, the maximal number of distinct port sets can be deduced, and so for the number of distinct routes (a route is uniquely defined by its associated port set). We obtain here 4 routes, from the one defining no data port (the associated port set is empty), to the one defining all possible data ports (the associated port set contains port 0 and port 1).

number of data messages:
We chose to modelise 2 distinct data messages, labelled from 0 to 1.

From these values, and with the environment we described in section 4.1, the CÆSAR-ALDÉBARAN compiler generates for the whole specification of the Transit-Node a LTS of about 120 000 states and 500 000 transitions. In the sequel, this LTS, representing the exhaustive behaviour of the Transit-Node described by the LOTOS specification will be noted S_{TN}.

It seems at this point that the choices we made offer a reasonable compromise: the specification is general enough to modelise non-trivial behaviour of the Transit-Node, and the size of the model obtained remains small enough to expect subsequent verifications to be carried out on a reasonable execution time.

5 Correctness of the specification

Once the Transit-Node system has been fully described, it remains to verify that the LOTOS program obtained is correct with respect to its initial informal description. In other words, we have to ensure that this LOTOS program can be considered as a *formal specification* of the Transit-Node allowing – for instance – to design an executable implementation.

In the following sections we check each clause of the informal definition of section 2, trying to establish whether it is preserved or node in the LOTOS description. More precisely, this set of clauses is split into three classes:

- the clauses which are preserved "by construction" of the LOTOS program,
- the clauses which require a verification at the LTS level,
- the clauses which are not preserved by the LOTOS program.

Each class is presented in turn, respectively in sections 6, 7 and 8.

6 Clauses preserved "by construction"

Most of the clauses concern only some very general features of the Transit-Node design. Therefore they are straightly preserved, by the program architecture itself.

clause 1:

Control Port-In, *Control Port-Out*, *Data Port-In* and *Data Port-Out* have been specified. Two data ports can be created ($N = 2$), and four distinct routes can be defined ($M = 4$).

clause 2:

(a) Each Port is serialised: they accept only one message at a time from the environment (*rendez-vous* mechanism).

(b) They are concurrent to each others: they are implemented as independent processes, without any synchronisation on a control or data message reception.

(c) A data message can be received on a data port `p` only when the corresponding `Inport` subprocess becomes active, which happens on reception of a *Create-Port(p)* internal message. Since this message is sent only when a port creation has been requested, the clause is preserved.

clause 5:

Initially, even if all LOTOS processes are created, only process `Controller` is active (i.e., may receive messages from the environment). Data ports are not active, since subprocesses `Inport` and `Outport` are not active.

clause 6:

This clause mainly refers to the behaviour of process `Controller`:

(a) Data ports can only be defined once (so only *new* ports are created). A data port creation (internal message *Create-Port(p)*) activates new `Inport` and `Outport` subprocesses, able to accept data messages (see clause 5).

(b) Routes can be created several times (a same route number `r` can define several successive port sets `s`). Whenever a route is (re)-defined, the data structure maintaining the current route set is update.

clause 8:

Processes `ErrHandler` and `Outport` accept any data messages, without restrictions on their value. Control messages are not sent directly to them, but through the internal message *Control-Error-In*. This message is itself generated when an erroneous control message reception occurs. Messages leaving the node through *Data Port-Out* are non deterministically chosen in the associated buffers, so any ordering is likely to happen.

Some of the "proof" arguments proposed here may seem tricky, or not sufficiently motivated. In fact, to be really rigorous, some of them would require deeper verifications, for instance at the LTS level. However, if such a check remains always possible, we prefer to keep this approach for more interesting clauses.

7 Clauses verified at the LTS level

A second class of clauses concerns more fundamental properties of the Transit-Node, related to its "run-time behaviour". In other words, it describes how any implementation of the node should behave, from a functional point of view. Consequently, these clauses cannot be established only by a source level analysis of the specification, and they require deeper verifications, carried out on the underlying model (which represents the exhaustive behaviour of the node).

We present these clauses in three parts, depending if they refer to the general behaviour of the node, to the routing of data messages or to the management of faulty messages. All these clauses are verified on the LTS S_{TN}, introduced in section 4.2.

7.1 General behaviour of the node

The expected behaviour of the node with respect to data messages is described by clause 4, saying that "all data messages will eventually transit the node or become faulty".

As it is often the case, the verification of this property is split into a "safety part" and a "liveness part".

7.1.1 Safety

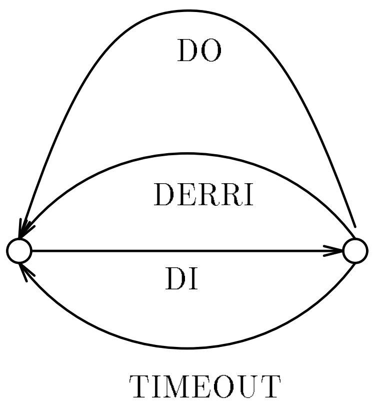
From a safety point of view, clause 4 can be rephrased as:

> Any received data message will have the ability to exit the node from a data port-out, or to become faulty.

This statement refers to four events in our LOTOS specification:

- data message reception, occurring on gate `DI` (through a *Data Port-In*)
- data message emission, occurring on gate `DO` (through a *Data Port-Out*)
- faulty data message occurrence, notified either on gate `DERRI` (*Data-Error-In* internal message) or on gate `TIMEOUT` (*Timeout* internal message).

Thus, these gates constitute the set V of *visible actions* we want to observe to check if the clause is preserved. Moreover, if we consider only V actions devoted to a single data message (for instance message 0, actions devoted to other messages being hidden), the previous statement can be represented by the LTS S_{4s}:



This LTS can be interpreted as follows:

- after its reception, data message 0 *can* always be sent out the node or be buffered as faulty
- one of this alternative *must* occur before the data message can re-enter the node.

Using the CÆSAR-ALDÉBARAN toolbox, it can be established that this LTS is *safety equivalent* to the one representing the exhaustive behaviour of the Transit-Node: $S_{TN} \approx_s S_{4s}$. The same verification can be performed with data message 1, and consequently the safety part of clause 4 is satisfied.

7.1.2 Liveness

It now remains to check that this expected behaviour is always executed, which means that no deadlocks nor livelocks has been hidden during the safety equivalence comparison. Using CÆSAR-ALDÉBARAN, this can be performed as follows :

- we compute the quotient S_{4l} of LTS S_{TN} with respect to *branching bisimulation*[4], when only V actions are visible,
- we check whether S_{4l} contains deadlock states,
- finally, we also check if LTS S_{TN} contains livelocks when only V actions are visible (since livelocks are not preserved by branching bisimulation, this check cannot be performed on S_{4l}).

Surprisingly, if no deadlock were found, this first experiment revealed the presence of numerous livelocks. In fact, these livelocks occur because our LOTOS specification is *unfair*. More precisely, according to our description (but not according to the informal one !), the Transit-Node may treat forever the same control message, provided it is offered infinitely often by the environment. Thus, at any time, the Transit-Node may try for instance to always redefine the same route, or to always proceed the same incorrect message: when these actions are hidden, such behaviours are expressed by livelocks in the resulting LTS.

Of course, we want to omit this undesirable behaviour in the verification process, in order to detect only the livelocks resulting from an incorrect specification (i.e., which happen because of internal communications). Consequently, the solution we adopted is to extend the set V of visible actions during the livelock detection by adding all control message receptions (i.e., actions occurring on gate `CI`). Thus, “expected” livelocks disappear, since they correspond now to visible actions. Clause 4 could then be verified in practice.

7.2 Data message routing

Inside the Transit-Node, the routing of data messages is described by clause 7, which asserts the following properties :

(a) when a data message `M` is received with a route indication `R`, then it is routed to any one of the open *Data Ports-Out* associated to `R` *at the time of its arrival*, or put in the faulty collection if no such port exists,

[4]In practice this quotient is not computed from LTS S_{TN}, but straightly from the LOTOS specification

(b) the *Data Ports-Out* are serialised, a buffer being associated with each port,

(c) a data message is faulty when its total transit time within the node becomes greater than a constant time T_c.

Part **(b)** is verified by construction of the LOTOS program, and consequently only parts **(a)** and **(c)** remain to be verified.

7.2.1 clause 7 (a)

We only need to verify a "safety" interpretation of the clause, since clause 4 already ensures that any data message leaves the node on a *Data Ports-Out*, or is put in the faulty collection. Moreover, as the management of faulty messages will be discussed in section 7.3, the property we want to check here can be rephrased as follows :

> Whenever a data message `M` has been received with a route indication `R`, and *if* this message is routed to a *Data Port-Out* `P`, then `P` must belong to the port set associated to route `R` at the time of the message arrival.

To verify this property, we need to determine the port set associated to a given route when a data message is received. In the LOTOS program, this information is represented by an internal variable of the `Controller` process. Therefore, it cannot be straightly accessed at the LTS level (since it is encoded inside the states). However, this information can be retrieved at any time by keeping track of each route definition received by the node (i.e., *Add-Route* messages).

Consequently, to verify this property for a given message `M`, received with route indication `R`, the relevant actions of the LOTOS program are the following:

- *message reception:* `DI !P !M !R`, which indicates the reception of `M` on the data port-in `P`,
- *route definition:* `CI !ADROUTE !R !PS`, which associates port set `PS` to route `R`
- *message delivery:*, `IO !P !M`, which indicates that `M` is put in the buffer associated to data port-out `P`.

Unfortunately, if we compute the quotient of LTS S_{TN} with respect to safety equivalence for this set of visible actions, we obtain a LTS of 60 states and 472 transitions, which is too large to conclude "by hand on this" LTS whether the property is verified or not.

Therefore, we adopted another approach :

1. We consider a general behaviour which express the expected property. This behaviour can be described for instance by a LOTOS program, and translated into a LTS S_{7a}.
2. We verify that the actual behaviour of the Transit-Node "is included" in this expected behaviour (i.e., that every visible action performed by the Transit-Node is "allowed" with respect to the property). Such a notion of inclusion can be expressed by the *safety preorder* relation: $S_{TN} \sqsubseteq_s S_{7a}$.

We give a first specification of a general behaviour expressing Clause 7 (a), for a given message `m` and a given route `r`. Intuitively, this behaviour is the following :

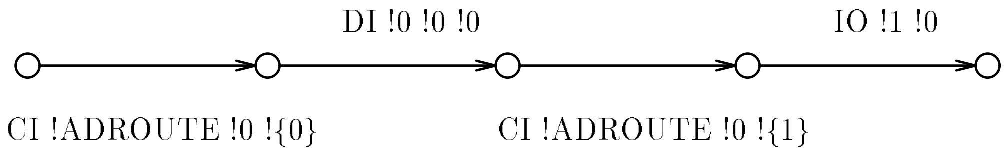
- A port set `current` (initially empty) maintains the definition of route `r`.
- After reception of message `m`, two scenarios are allowed:
  - either `m` is already faulty, so it will not be delivered on a data port-out (and there is nothing to verify),
  - or, `m` is correct, and the expected behaviour is that `m` will be delivered on a data port-out belonging to `current`. However, as route `r` can be redefined at any time before `m` is delivered, the current value of set `current` has to be memorised (variable `ps0`). Finally, `m` may also become faulty at any time (e.g., timed-out), and therefore not be delivered.

We give a LOTOS description of this behaviour :

```
process Waiting [CI, DI, IO]
        (m:Env, r:RouteNo, current:PortSet) : noexit :=

  (CI !ADD_ROUTE !r ?ps:PortSet ;
  (* re-definition of route r : set "current" is updated *)
                               Waiting [CI, DI, IO] (m, r, ps))
[]
  (DI ?p:PortNo !m ?r0:RouteNo;
        (* reception of data message m with route indication r0 *)
    (
     Idle [CI, DI, IO] (m, r, r0, current, current)
     (* message m may be correct, it will be routed on a data port *)
    []
     Waiting [CI, DI, IO] (m, r, current)
     (* message m is already faulty, it won't be routed on a data port *)
   )
  )
endproc

process Idle [CI, DI, IO]
(m:Env, r:RouteNo, r0:RouteNo, current:PortSet, ps0:PortSet) : noexit :=

 (IO ?p:PortNo !m [(r0 eq r) and (p IsIn ps0)] ;
   (* message m has been received on route r :
      it must be routed on a data port belonging to set ps0 *)
                      Waiting [CI, DI, IO] (m, r, current))
[]
 (IO ?p:PortNo !m [not (r0 eq r)] ;
   (* message m has been received on a route different than r :
        no verification is performed *)
```

```
                         Waiting [CI, DI, IO] (m, r, current))
[]
 (CI !ADD_ROUTE !0 of RouteNo ?ps:PortSet ;
   (* re-definition of route 0 : set "current" is updated *)
                        Idle [CI, DI, IO] (m, r, r0, ps, ps0))
[]
 Waiting [CI, DI, IO] (m, r, current)
   (* message m happens to be a faulty one,
        it will not be routed on a data port *)
endproc
```

Surprisingly, it appears in practice that our description of the Transit-Node does not verify this specification. More precisely, the diagnostic message returned by ALDÉBARAN indicates that the following execution sequence can be performed by the Transit-Node, whereas it is not allowed by the above specification:



This unexpected scenario is:

1. route 0 is defined a first time, associated to port set {0} ;
2. data message 0 is received, with route indication 0 ;
3. route 0 is then redefined, associated to port set {1} ;
4. finally, data message 0 is delivered on data port-out 1, instead of the expected port-out 0 (associated to route 0 at the time of the message arrival).

In fact, the reason why such a sequence appears in the LOTOS specification is twofold :

- when a data message is received, deciding on which data port-out it can be routed is not an *atomic* action: internal messages have to be exchanged between `DataInPorts` and `Controller` processes.
- the ports are independent and concurrent each other: a route can be redefined at any time on *Control Port-In*, even while process `DataInPorts` is handling a data message.

Consequently, when a route is redefined just after the reception of a data message, it is impossible to determine which definition of the route will be taken into account during the routing operation. Note that this behaviour corresponds in practice to the one observed from outside the Transit-Node, and is allowed by its original informal description.

According to this first experiment, a weaker specification has to be proposed to express clause 7 (a). We give here such a specification, corresponding to a new interpretation of this clause, properly dealing with route redefinition :

> Whenever a data message M has been received with a route indication R, and *if* this message is routed to a *Data Port-Out* P, then P must belong to the port set associated to route R *at any time between the message arrival and its delivery.*

Thus, process Idle is modified by adding to port set ps0 the new ports associated to each redefinition of route R :

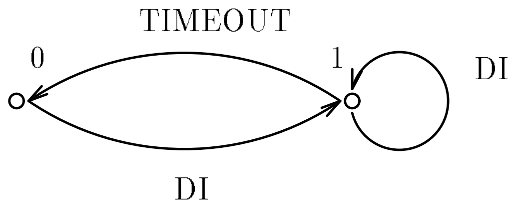
```
process Idle [CI, DI, IO]
(m:Env, r:RouteNo, r0:RouteNo, current:PortSet, ps0:PortSet) : noexit :=

 (IO ?p:PortNo !m [(r0 eq r) and (p IsIn ps0)] ;
   (* message m has been received on route r :
      it must be routed on a data port belonging to set "current"  *)
                        Waiting [CI, DI, IO] (m, r, current))
[]
 (IO ?p:PortNo !m [not (r0 eq r)] ;
   (* message m has been received on a route different than r :
      no verification is performed *)
                        Waiting [CI, DI, IO] (m, r, current))
[]
 (CI !ADD_ROUTE !0 of RouteNo ?ps:PortSet ;
   (* re-definition of route 0 : sets "ps0" and "current" are updated *)
                        Idle [CI, DI, IO] (m, r, r0, ps, Union(ps0, ps)))
[]
 Waiting [CI, DI, IO] (m, r, current)
   (* message e may happen to be a faulty one,
      it will not be routed on a data port *)
endproc
```

This new specification is verified by the LOTOS description of the Transit-Node: $S_{TN} \sqsubseteq_s S_{7a}$.

7.2.2 clause 7 (c)

As we do not quantify in the LOTOS specification the transit time of data messages within the node, the only point remaining to be checked with respect to clause 7 (c) is that a (correct) data message received by the node can always become faulty because a timeout occurs before its delivery. In other words, such a message cannot be prevented to become "timed-out".

For a given message M, this safety property can be expressed by the following LTS S_{7c}:



where:

- label `DI` denotes the reception of message `M` through a *Data Ports-In* (action `DI !P !M !R` in the LOTOS specification, where `P` and `R` are any port and route identifier),
- label `TIMEOUT` denotes a timeout for message `M` (action `TIMEOUT !M` in the LOTOS specification)

The intuitive meaning of this LTS is that whenever message `M` is inside the node (state 1), then it may become faulty because of a timeout event.

Using CÆSAR-ALDÉBARAN we can show that this LTS is safety equivalent to the one modeling the LOTOS specification ($S_{TN} \approx_s S_{7c}$) when only `DI` and `TIMEOUT` actions are visible, which ensures that clause 7 (c) is preserved by the LOTOS specification.

7.3 Faulty messages management

The expected behaviour of the Transit-Node with respect to faulty messages is described by several clauses of its informal specification:

- clause 10 defines the exact meaning of "faulty messages": they are either undefined control messages, or data messages with an incorrect route indication, or timed-out data messages,
- clause 9 indicates that all faulty messages have to be stored in a *faulty message collection* until reception of a *Send-Faults* control message,
- and finally clause 6 (c) and 11 (c) indicate that a succession of *Send-Faults* messages must eventually route all faulty messages outside the node (through *Control Port-Out*), in any order.

We examine successively these two phases, namely faulty messages collection (clause 9 and 10), and faulty messages delivery (clause 6 and 11).

7.3.1 Faulty messages collection

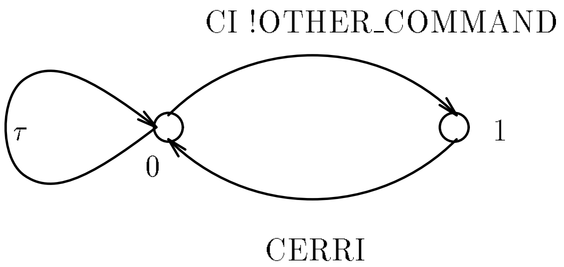
In the LOTOS description, the bufferisation of each possible faulty message (as defined by clause 10) corresponds to communications with the `ErrHandler process`:

- `CERRI` action for an undefined control message (*Control-Error-In* internal message),
- `DERRI` action for an incorrect data message (*Data-Error-In* internal message),
- `TIMEOUT` action for a timed-out data message (*Timeout* internal message).

We verify that, whenever a faulty message is detected, one of these actions is eventually executed:

undefined control message reception:
They correspond to an *Other-Command* message reception by the *Control Port-In*. Therefore it is sufficient to show that any occurrence of this message is followed by a `CERRI` action. This check can be done using *branching bisimulation*, followed by a livelock detection: using CÆSAR-ALDÉBARAN, we compute the quotient of LTS S_{TN} for branching bisimulation,

when only these two actions are visible, together with a *divergent predicate* indicating if each state of the quotient corresponds or not to a livelock in the LOTOS program. We obtain the following LTS, where only the initial state 0 is divergent:



Consequently, all incorrect messages are eventually bufferised.

incorrect data message reception:

A data message can be incorrect because its associated route is either undefined, or refers to an empty open port set. According to clause 4, to show that such messages are eventually bufferised as faulty, it is sufficient to show that they *cannot* be delivered on a *Data Port-Out*. To perform this check using CÆSAR-ALDÉBARAN we proceed as follows:

1. We modify the environment in such a way that a given data message (lets say data message 0) is always faulty. For instance, this message is always associated to an undefined route, or its associated route always refers to a set of non open ports.
2. We then show that, within this environment, message 0 cannot be delivered on a data port. This can be done by computing the quotient of S_{TN} with respect to *safety equivalence*, when only action `DO` dedicated to message 0 are kept visible. As expected, this LTS contains no transition.

It was not possible here to verify the clause in a more general context (i.e., without modifying the environment). In fact, as we already seen, it is not possible to precisely determine the route status associated to a data message – and consequently if this message is faulty or not – when considering only external actions of the Transit-Node. Then, the solution we adopted was to modify (temporarily) the environment in order to force a message to become faulty.

timed-out data message:

Timed-out messages are always bufferised, since in our description, the same internal message *Timeout* both indicates a timed-out message occurrence and sends it to the `ErrHandler` process.

7.3.2 Faulty messages delivery

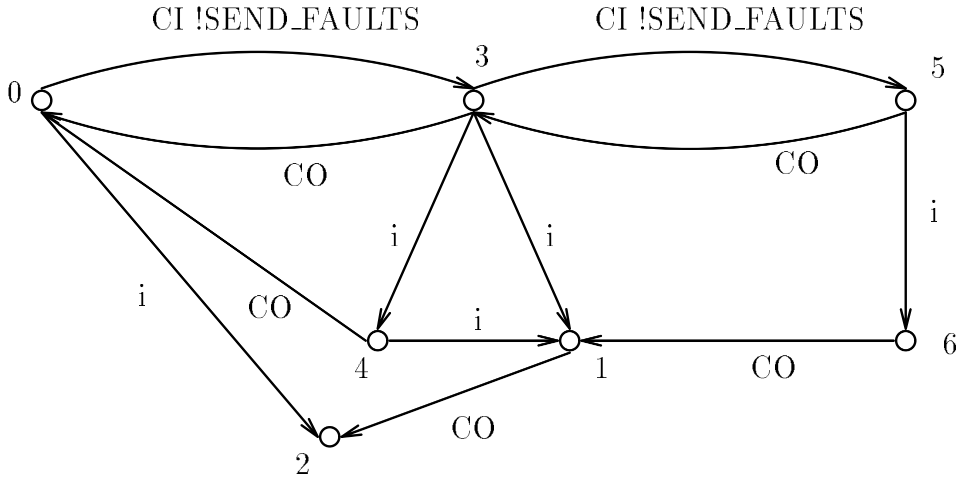
The expected behaviour of the Transit-Node with respect to faulty message delivery is defined by clause 11 and 6 (c). They indicate that, on a *Send-Faults* message reception (*resp.* on a succession of such receptions), *some of* (*resp. all*) the faulty messages bufferised so far must be sent outside the node, through the *Control Port-Out*.

Here again, the verification is performed in two steps :

1. We first show that each *Send-Faults* request is always followed by an emission through *Control Port-Out*. This check allows to verify the liveness part of the requirement.

2. Then, we verify that messages emitted on *Control Port-Out* are exactly messages previously bufferised as faulty. Thus, the safety part of the requirement is preserved too.

To verify the liveness part within CÆSAR-ALDÉBARAN, we compute the quotient of S_{TN} with respect to *branching bisimulation* when only `CI !SEND_FAULTS` and `CO` actions are visible (corresponding respectively to a *Send-Faults* request and a *Control Port-Out* emission). The resulting LTS is the following :



This quotient is small enough to check that, for all execution sequences, each occurrence of a `CI !SEND_FAULTS` action is eventually followed (later in the sequence) by an occurrence of a `CO` action. Note that this quotient contains a sink state (state 2), since, according to our environment, at any time the Transit-Node may stop to receive any *Send-Faults* message.

Finally, we also performed a second check to verify that only "expected" livelocks occur when these two actions are visible. Consequently, each *Send-Faults* request is always followed by a *Control Port-Out* emission.

To simplify the verification of the safety part of the requirement, we consider successively control messages case, and data messages case.

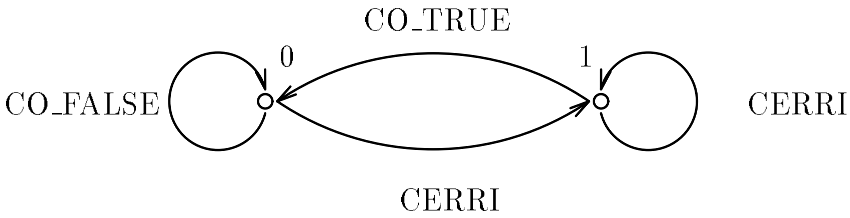
1. Control messages

The requirement we have to check is that a control message is emitted on *Control Port-Out* if and only if it has been previously bufferised by a *Control-Error-In* internal message. In the LOTOS program, these two events are respectively expressed by a `CO !TRUE !L` action (for any value `L` of sort `EnvList`), and by a `CERRI` action. So, this requirement can now be rephrased in terms of execution sequences of the LTS S_{TN} :

- a `CERRI` action cannot be followed by a `CO !FALSE !L` action (i.e., each control message bufferised as faulty must leave the node upon request),

- two successive occurrences of a `CO !TRUE !L` action must be separated by an occurrence of a `CERRI` action (i.e., only faulty control messages can leave the node through *Control Port-Out*).

In practice these two properties are verified, since, when only `CERRI` and `CO` actions are kept visible, and when each `CO !TRUE !L` (*resp.* `CO !FALSE !L`) action has been renamed in `CO_TRUE` (*resp.* `CO_FALSE`), then LTS S_{TN} is *safety equivalent* to the following one (where state 0 is the initial state):



2. Data messages

It remains to check that a data message is emitted on *Control Port-Out* if and only if it has been previously bufferised by either a *Data-Error-In* or a *Timeout* internal message. In the LOTOS program, these events are expressed as follows:

- an emission of a data message `M` on *Control Port-Out* is represented by a `CO !B !L` message, where `B` can be any boolean value and `L` is a list of message, containing `M`,
- for a given data message `M`, internal messages *Data-Error-In* and *Timeout* are represented respectively by `DERRI !M` and `TIMEOUT !M` actions.

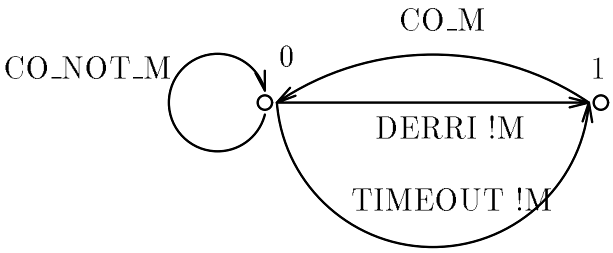
Here again, the requirements can be rephrased in terms of execution sequences of the LTS S_{TN}:

- after a `DERRI !M` or a `TIMEOUT !M` action, the next occurrence of `CO !B !L` action must be such that `M IsIn L` (i.e., each data message bufferised as faulty must leave the node upon request),
- two successive occurrences of a `CO !B !L` action, such that a given `M` belongs to `L`, must be separated by an occurrence of either `DERRI !M` action or a `TIMEOUT !M` action (i.e., only faulty data messages can leave the node through *Control Port-Out*).

As for control messages, we can show using CÆSAR-ALDÉBARAN that for each data message `M`, these two properties are verified: if,

1. only `DERRI !M`, `TIMEOUT !M` and `CO !B !L` actions are visible,
2. each `CO !B !L` action such that `M IsIn L` (*resp.* `not (M IsIn L)`) is renamed in `CO_M` (*resp.* in `CO_NOT_M` action),

then the LTS S is *safety equivalent* to the following one (where state 0 is the initial state):



8 Non preserved clauses

Finally, some of the clauses present appearing in the informal description of the Transit-Node are clearly *not* preserved by our formal specification, either because of the choices we made, or because the formalism we used was not suitable enough.

Clause 3:

As we mention in the previous section, our specification does not describe a *fair* Transit-Node, in the sense that some messages may be ignored forever. More precisely, this notion of fairness appears at different steps of the specification:

- All data or control messages proposed infinitely often by the environment must be accepted by the node, and no data port should benefit from any priority. This statement is not verified in our specification, since the *same* message can be accepted forever, and then starving the other ones.
- Similarly, all internal messages must be fairly treated by the processes modeling the Transit-Node. For instance, no route status request addressed to the `Controller` process by a given `OutPort` subprocess (*Route-Query* internal message) should be postponed forever. Here again, this is not true in our specification.
- For a data message emission, the choice of a data port-out among the set of ports defined by its associated route must be fair. No port should have any priority. This condition is verified in our specification, since it describes all possible choices.
- No data message can stay forever on the buffer associated to a data port-out, nor in the faulty message buffer. This statement is verified in our specification : port-out buffers are managed as fifo queues, and the faulty buffer is flushed by an atomic action.

However, to verify liveness properties, we assumed a fair behaviour of the Transit-Node by rejecting unfair execution sequences at the model level. Another solution would have been to prevent these sequences to occur at the program level, for instance by adding explicit schedulers on critical message receptions.

Clause 11 (a) and Clause 12:

Since we do not take into account timing aspects on a quantitative point of view these two clauses are not relevant in our specification.

Conclusion

We have presented in this report a LOTOS specification of a Transit-Node informal description, together with its formal verification following a "model-based" approach. From this practical experiment, several comments can be done with respect to the general method we used.

First of all, the formalism we consider to formally specify the Transit-Node greatly influenced the description we obtained. More precisely, choosing a "process algebra oriented" formalism like LOTOS naturally leads to an operational description of the system (answering to "how is it built ?" rather than to "what must it perform"). This is not without consequences in the verification process, since properties which are checked relies on this system description, and hence must be expressed on the same vocabulary, and following a similar point of view. Therefore, such formalisms seems more suitable to specify rather detailed system descriptions, including design considerations, whereas more declarative formalisms (such abstract data types) are more adapted to deal with higher-level descriptions. Thus, these two kinds of formalisms should be both used in a verification process, at different stages, depending on the abstraction level of the system description under consideration.

Furthermore, one can also notice that a formalism based on communicating processes allows to easily describe system environments (the environment being specified as an external and independent process, progressing in parallel with the system). However, this technique also suffers from some limitations, since (for instance) it hardly allows to describe *fair* environments.

Regarding the verification method, one the most important characteristic of model-based approaches is that, within their application domain, they can be considered as *exhaustive* and *decidable*. Moreover, and even if current results are not yet completely satisfying, verification algorithms and tools are now able to deal with rather large models, corresponding to non-trivial system descriptions.

However, expressing the correct properties to verify may remain difficult, and sometimes tricky. This aspect was enforced in this case-study, since the toolbox used did not provide (yet) any temporal logic checker, and each property has been expressed as a behavioural specification. A related problem is that the LTS generated from the LOTOS program does not contain any more references to the *variables* appearing in this program (for instance the "route definitions" in the Transit-Node). Consequently, properties which can be checked on this LTS cannot refer to these variables, and must be expressed only in terms of *visible actions* of the LOTOS program. Even if this constraint corresponds to something realistic in practice, since the expected behaviour of the program is usually defined with respect to an external observer (who cannot access its internal variables), it could be sometimes useful being able to refer to such variables in the properties to verify, at least for debugging purpose (in the first stage of the verification process, or to investigate why a given specification is not true).

Finally, further work could be carried out on this case study, for instance dealing in a more quantitative point of view with timing considerations, or extending the correctness proofs for arbitrary values of the system parameters (i.e., the number data ports, of distinct routes, or distinct data messages, ..., etc.). This would require more powerful specification formalisms and verification techniques.

References

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. *Safety for Branching Time Semantics*. In *Proceedings of the 18th ICALP, Madrid, Spain*, volume 510 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, July 1991.

[FGM+92] Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodríguez, and Joseph Sifakis. A Toolbox for the Verification of LOTOS Programs. In Lori A. Clarke, editor, *Proceedings of the 14th International Conference on Soft-*

ware Engineering ICSE'14 (Melbourne, Australia), pages 246–259, New-York, May 1992. ACM.

[FKM93] J.C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, 1993.

[FM91] Jean-Claude Fernandez and Laurent Mounier. "On the Fly" Verification of Behavioural Equivalences and Preorders. In K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, volume 575 of *Lecture Notes in Computer Science*, Berlin, July 1991. Springer Verlag.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, Amsterdam, June 1990. IFIP, North-Holland.

[GW89] R.J van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[ISO87] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, July 1987.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer Verlag, Berlin, 1980.

[NV90] Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. In *Proceedings 5th Annual Symposium on Logic in Computer Science (LICS 90), Philadelphia USA*, pages 118–129, Los Alamitos, CA, June 1990. IEEE Computer Society Press.

[Par81] David Park. *Concurrency and Automata on Infinite Sequences.* In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, Berlin, March 1981.

Appendix: abstract data types

We give here the set of abstract data types used in the LOTOS specification of the Transit-Node build upon the standard Boolean and Integer libraries.

```
type PORT is NATURAL renamedby
   sortnames PortNo for Nat
endtype

type ROUTE is NATURAL renamedby
   sortnames RouteNo for Nat
endtype

type ENVELOPPE is NATURAL renamedby
   sortnames Env for Nat
endtype

type COMMANDS is
  sorts Command
  opns
        Add_Data_Port (*! constructor *),
        Add_Route (*! constructor *),
        Send_Faults (*! constructor *),
        Other_Command (*! constructor *)
                : -> Command
endtype

type ERROR_CODE is
  sorts ErrorCode
  opns
        Unknown_Route (*! constructor *),
        (* Timed_Out constructor ,*)
        Wrong_Msg (*! constructor *)
                : -> ErrorCode
endtype

type PORT_SET is BOOLEAN, PORT
  sorts PortSet
  opns
        emptyset (*! constructor *) : -> PortSet
        add (*! constructor *) : PortNo, PortSet -> PortSet

        _IsIn_ : PortNo, PortSet -> Bool
        _includes_ : PortSet, PortSet -> Bool
        _==_ : PortSet, PortSet -> Bool

  eqns
  forall ps, ps1, ps2:PortSet, p, p1, p2:PortNo
    ofsort Bool
        p IsIn emptyset = false ;
        p IsIn add(p, ps) = true ;
```

```
        not (p1 eq p2) =>  p1 IsIn add(p2, ps) = p1 IsIn ps ;
    ofsort Bool
        ps includes emptyset = true ;
        not (p IsIn ps1) => ps1 includes add(p, ps2) = false ;
        p IsIn ps1 => ps1 includes add(p, ps2) = ps1 includes ps2 ;
    ofsort Bool
        emptyset == emptyset = true ;
        emptyset == add(p2, ps2) = false ;
        add(p1, ps1) == emptyset = false ;
        add(p1, ps1) == add(p2, ps2) = (p1 eq p2) and (ps1 == ps2) ;
endtype

type ROUTE_LIST is BOOLEAN, PORT_SET, ROUTE
  sorts RouteList
  opns
        emptyrl (*! constructor *) : -> RouteList
        insert (*! constructor *)  : RouteNo, PortSet, RouteList -> RouteList

        _IsIn_ : RouteNo, RouteList -> Bool
        route  : RouteNo, RouteList -> PortSet
        update : RouteNo, PortSet, RouteList -> RouteList
  eqns
  forall rl:RouteList, r, r1, r2:RouteNo, ps, ps1, ps2:PortSet
     ofsort Bool
        r IsIn emptyrl = false ;
        r IsIn insert (r, ps, rl) = true ;
        not (r1 eq r2) => r1 IsIn insert(r2, ps, rl) = r1 IsIn rl ;
     ofsort PortSet
        route(r, emptyrl) = emptyset ;
        route(r, insert(r, ps, rl)) = ps ;
        not (r1 eq r2) => route(r1, insert(r2, ps, rl)) = route(r1, rl) ;
     ofsort RouteList
        update(r, ps, emptyrl) = emptyrl ;
        update(r, ps1, insert(r, ps2, rl)) = insert(r, ps1, rl) ;
        not (r1 eq r2) =>
                update(r1, ps1, insert(r2, ps2, rl)) =
                                        insert(r2, ps2, update(r1, ps1, rl)) ;
endtype

type ENV_LIST is BOOLEAN, ENVELOPPE
  sorts EnvList
  opns
        emptyl (*! constructor *) : -> EnvList
        insert (*! constructor *) : Env, EnvList -> EnvList

        head : EnvList -> Env
        tail : EnvList -> EnvList
        remove : Env, EnvList -> EnvList
        _IsIn_ : Env, EnvList -> Bool
        _==_ : EnvList, EnvList -> Bool

  eqns
  forall l, l1, l2:EnvList, r, r1, r2:Env
```

```
    ofsort Env
        head(insert(r,l)) = r ;
    ofsort EnvList
        tail(insert(r,l)) = l ;
        remove(r, emptyl) = emptyl ;
        remove(r, insert(r, l)) = l ;
        not (r1 eq r2) =>
                remove(r1, insert(r2, l)) = insert(r2, remove(r1, l))  ;
     ofsort Bool
        r IsIn emptyl = false ;
        r IsIn insert (r, l) = true ;
        not (r1 eq r2) => r1 IsIn insert(r2, l) = r1 IsIn l ;
    ofsort Bool
        emptyl == emptyl = true ;
        emptyl == insert(r2, l2) = false ;
        insert(r1, l1) == emptyl = false ;
        insert(r1, l1) == insert(r2, l2) = (r1 eq r2) and (l1 == l2) ;
endtype
```