Concurrency and Real-time Specification with Many-Sorted Logic and Abstract Data Types: an Example

Teodor Knapik
IREMIA, Université de la Réunion
BP 7151
97715 SAINT DENIS Messag. Cedex 9 France
e-mail: knapik@univ–reunion.fr

Abstract

We discuss the use of algebraic specifications for the description of the requirements of concurrent and real time systems. The underlying logic is the usual Many-Sorted First Order Predicate Calculus with Equality without any concurrent features. In order to express dynamic and real time properties, we specify a data type, the role of which is to model time. This discussion is motivated by our specification of the "Transit Node" system.

1 Introduction

The first stage of formal system development consists of writing a *requirement specification*. It is a document that expresses abstractly properties of a system to be developed. Consequently, specification techniques used for this aim should provide great expressive power. Abstract data type specifications based on first order logic are widely recognized as such. However, the commonly-held opinion is that it is impossible to express concurrent and real time properties using this technique. This leads to the development of more specialized techniques on top of algebraic specifications. In [1] these are referred to as *algebraic specifications of concurrency*.

The aim of the Transit Node case study presented in this paper is to show that concurrent and real-time systems can be specified in a satisfactory way in the framework of "pure" algebraic specifications based on classical Many-sorted First Order Logic with Equality. This may be achieved by specifying an abstract data type, the rôle of which is to represent time. It is then possible to test the elapsed time and therefore express real time properties. We are also able to express concurrent properties. In order to say that two actions a and a′ must synchronize, we write $\forall t\ a(t) \iff a'(t)$. Both actions are represented by predicates a, a' : Time on the sort Time.

This paper is organized as follows. Section 2 provides the informal specification of the Transit Node system. In Section 3 we comment on an algebraic specification of this system. Section 4 is a discussion of the conformity of the presented algebraic specification with the informal specification. The last section is devoted to some conclusions and perspectives of this work.

2 Informal Specification

This case study was 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.

The informal specification reads as follows:[1]

Clause 1 *The system to be specified consists of a transit node with: one* Control Port-In, *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 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 starts 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 some messages in the faulty collection, if any, to* Control Port-Out. *The order in which the faulty messages are transmitted is not specified.*

[1] We present a slightly modified version of this specification with respect to the original one where some requirements turn out to be inconsistent.

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 (nondeterminate) of the open* Data Ports-Out *associated with route* m *at the time of arrival. If there is no such port the message becomes faulty.* **(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 transmission 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 eventually placed in the faulty collection where they stay until a* Send-Faults *command message causes them to be routed to* Control Port-Out.

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 **(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 to not 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 the faulty 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 with the specification.*

2.1 Modifications and Additional Assumptions

For the sake of simplicity, our formal specification will not completely conform to the informal requirements, precisely in the following points:

1. *All data ports are closed in the beginning if there is one. This differs from Clause* **5**.

2. *The buffering is unspecified although data messages cannot disappear inside of the TN. This slightly differs form Clause* **7b**.

We also consider the following additional assumptions:

Assumption 1 *The reception and the decoding of any message may take some time which is smaller than the constant* T.

Assumption 2 *The information carried by a control message is always correct. This means that a control message may not attempt to open a non-existent port, define a non-existent route or associate a non-existent port to a route.*

Assumption 3 *The routing information carried by a data message is always correct.*

Assumption 4 *A message cannot arrive twice at the TN. In fact, nothing is assumed about equality of messages. Thus in the models of this specification, different occurrences of a message are represented by different messages. An additional abstraction may be gained using an observational equality which is not a congruence as in* [5]. *This would allow to identify two messages which arrive at different instants*

Assumption 5 *All routes are empty in the beginning if there is one.*

Assumption 6 *Both control port-in and control port-out are always open.*

Assumption 7 *Any transmission of a correct data message will end before the total transit time of the message through the TN becomes greater than a constant* T.

3 Formal Specification

In this section we describe our formal specification of the Transit Node. This is written in the PLUSS specification language [3]. Note that all free variables occurring in axioms are considered as implicitly universally quantified.

3.1 Specifying Time

As mentioned in introduction, it appears essential to have a model of time on the top of which our specification could be built. For this reason, we provide a specification of time which describes the models we are interested in. Note that this is not required by the informal specification.

According to Clause **12**, a source of time is available in the environment. This leads us to the simplification that each component if the TN is evolving in the same global time. Consequently, we try to describe a class of linear models of time which includes both discrete and dense models, with or without an initial instant.

spec : TIME **sort :** Time **operations :** $\_ + \_$: Time Time $\rightarrow$ Time **predicates :** $<, \leq$: Time Time **axioms :** $\neg t < t$, $(t < t' \wedge t' < t'') \Rightarrow t < t''$, $t \leq t' \iff (t < t' \vee t = t')$, $t \leq t' \vee t' \leq t$, $t + t' = t' + t$, $(t + t') + t'' = t + (t' + t'')$, $t < t' \iff t + t'' < t' + t''$ **where :** t, t', t'' : Time **end** TIME.	**spec :** TIME_WITH_T **use :** TIME **sort :** Time **operations :** T : $\rightarrow$ Time **end** TIME_WITH_T.

Figure 1: Time

We argue that the specification of the time of Figure 1 is abstract enough in the sense that it describes the models we are interested in. The time is defined by means of an ordering and a sum. The 4 first axioms define a strict total ordering and the associated non-strict ordering. The sum is defined as an associative-commutative operation growing in each of its arguments. It is possible to slightly modify this specification in order to deal only with dense time. This may be achieved by adding the axiom:

$$\forall t \; \forall t'' \; \exists t' \quad t < t'' \Rightarrow (t < t' \wedge t' < t'')$$

It also easy to restrict to models with an initial instant by introducing a constant $0 :\rightarrow$ Time and adding the axioms $0 \leq t$ and $t + 0 = t$.

Notice that in the framework of total algebras, specification TIME has only infinite models. This is due to the fact that the sum is a growing and total operation. A slightly more abstract specification, including both finite and infinite models, would be obtained in the framework of partial algebras which has not been used here for sake of simplicity.

As specified in TIME_WITH_T (see Figure 1) module TIME is enriched with the constant T stipulated in Clauses **7c**, **10c** et **11a**.

The problem of specifying time is tackled here as an example. A deeper study of this point is necessary.

3.2 Data Ports and Routes

In order to define data ports and routes, we introduce in module CONST (see Figure 2) constants N et M which express the number of data ports and routes.

spec : ROUTE_INDEX
  use : CONST
sort : Route
operations :
  route_num : Route → Nat
axioms :
  $m < M \Rightarrow \exists r\ route\_num(r) = m,$
  $route\_num(r) = route\_num(r') \Rightarrow r = r'$
  where : r, r': Route; m: Nat
forgets :
  route_num, M, Nat
end ROUTE_INDEX.
	spec : CONST
	  use : NAT
	operations :
	  N : → Nat
	  M : → Nat
	axioms :
	  $0 < N,$
	  $0 < M$
	end CONST.

Figure 2: Routes as a set of cardinality M

Sort Route is defined as a set of cardinality M. For this aim we consider an operation route_num : Route → Nat defined as an injective map whose range is the segment $[0, M-1]$. Due to the hierarchic constraints and "forgets" clause, specification ROUTE_INDEX has only one class of isomorphic models. Data ports

```
spec : PORT_INDEX
      use : CONST
sort : Port
operations :
      port_num : Port → Nat
axioms :
      n < N   ⇒   ∃p port_num(p) = n,
      port_num(p) = port_num(p')   ⇒   p = p'
      where : p, p': Port ; n: Nat
forgets :
      port_num, N, Nat
end PORT_INDEX.
```

Figure 3: Data ports as a set of cardinality N

are specified in an analogous way (see Figure 3). However it is important to notice that in our specification a data port represents a pair of ports: a data port-in and a data port-out. Due to Clause **6a** such a pair needs not to be represented by two separate entities since our specification preserves the independence of the reception and transmission of messages at an arbitrary given instant.

3.3 Control Messages

Specification module CTRL (see Figure 4) defines control messages and their arrivals to the TN. Predicates is_add_port, is_add_route, is_send_faults define different kinds of control messages stipulated in Clause **6**. According to Axioms 1–3,

spec : CTRL
    use : TIME_WITH_T
sort : Ctrl
operations :
    arrival, reception : Ctrl $\rightarrow$ Time
predicates :
    is_add_port, is_add_route, is_send_faults, is_unrecognized : Ctrl
    is_entering : Ctrl Time
axioms :
  1 : $\neg$(is_add_port(c) $\wedge$ is_add_route(c)),
  2 : $\neg$(is_add_port(c) $\wedge$ is_send_faults(c)),
  3 : $\neg$(is_add_route(c) $\wedge$ is_send_faults(c)),
  4 : unrecognized(c) $\Longleftrightarrow$ $\neg$(is_add_port(c) $\vee$ is_add_route(c) $\vee$ is_send_faults(c)),
  5 : arrival(c) $\leq$ reception(c),
  6 : is_entering(c, t) $\wedge$ is_entering(c′, t) $\Rightarrow$ c = c′,
  7 : is_entering(c, t) $\Longleftrightarrow$ arrival(c) $\leq$ t $\wedge$ t $\leq$ reception(c)
    where : c, c′ : Ctrl; t, t′ : Time;
end CTRL.

Figure 4: Control messages

any control message can have at most one of those 3 types. If none of the above predicates is satisfied, the message is unrecognized (see Axiom 4).

Predicate is_entering defines the activity of the control port-in. An atomic formula is_entering(c, t) is given the following meaning: *at time* t, *the control port-in is busy due to the reception of the control message* c. For this aim, each control message c is provided with the instant of its arrival to the TN (arrival(c)) and the instant when it is completely received by the TN and decoded (reception(c)). According to Axiom 7, the control port-in is occupied by the control message c between the instants arrival(c) and reception(c). The mutual exclusion (see Clause **2a**) on the control port-in is expressed by Axiom 6. Note that arrivals of data messages are specified in an analogous way.

3.4 Opening Data Ports

Specification module OPENING_PORT (see Figure 5) tells us how control mes-

```
spec : OPENING_PORT_INDEX
       use : PORT, CTRL
operations :
       port : Ctrl → Port
predicates :
       is_open : Port Time
axioms :
       ∀p ∀t (is_open(p, t)  ⟺  ∃c (reception(c) ≤ t ∧ is_add_port(c) ∧ port(c) = p))
       where :  p : Port;  t : Time;
end OPENING_PORT.
```

Figure 5: Action of control messages on data ports

sages act on data ports.

Operation port associates a data port with each control message. The entry of a control message c of type is_add_port into the TN causes the opening of the port port(c). Once more, the use of partial algebras would be more convenient (but more complicated), since the operation port needs not to be defined on messages, the type of which is not is_add_port.

The only axiom of this module states that a data port is open if and only if a control message has ordered its opening. It follows from the above that no data port is open before the reception of a control message. In particular, unlike in Clause 5 but according to our modification (see Section 2.1), at the initial state, if there is one, all data ports are closed.

3.5 Defining Routes

According to Clause 6b a route is defined by associating a set of data ports with it. For this reason, we introduce a specification of sets of data ports SET_OF_PORTS. The latter is obtained as an instance of the generic specification SET (assumed well known). The parameter ELEM is instantiated by the specification PORT_INDEX via the signature morphism Elem $\mapsto$ Port. The resulting sort Set is renamed into Ports.

Specification module DEFINING_ROUTE (see Figure 7) describes how control messages act on routes. Two additional operations[2] associate a route (def_route(c)) and a set of data ports (ports(c)) with each control message c. Operation

[2] Once again, these operations could more usefully be partial, which the total algebra framework does not allow.

```
spec :    SET_OF_PORTS
          as : SET(ELEM ↦ PORT_INDEX by Elem ↦ Port)
          renaming : Set into Ports
end SET_OF_PORTS.
```

Figure 6: Sets of data ports as an instance of the generic SET by PORT_INDEX

associated_ports(r, t) returns a set of data ports associated with a route r at time t.

At the reception of a control message of type is_add_route, the route def_route(c) has the set ports(c) associated with it. This is guaranteed by Axiom 2. Furthermore the set of data ports associated with a route can change at an instant t' with respect to an earlier instant t only if a control message of type is_add_route associates another set of data ports to this route at an instant bounded by t and t'. That is the meaning of Axiom 3.

According to Axiom 1 the set of data ports associated with a route is empty before receiving a control message that defines this route. In particular, at the initial state, if there is one, all routes are empty. This is in conformance with our Assumption 5.

```
spec : DEFINING_ROUTE
       use : ROUTE_INDEX, CTRL, SET_OF_PORTS
operations :
       def_route : Ctrl → Route
       ports : Ctrl → Ports
       associated_ports, open_ports : Route Time → Ports
axioms :
   1 : (∀c ((def_route(c) ∧ route(c) = r) ⇒ t < reception(c)))
          ⇒ associated_ports(r, t) = ∅,
   2 : is_add_route(c) ∧ def_route(c) = r
          ⇒ associated_ports(r, reception(c)) = ports(c),
   3 : (t < t' ∧ associated_ports(r, t) ≠ pts ∧ associated_ports(r, t') = pts
          ⇒   ∃c (t < reception(c) ∧ reception(c) ≤ t' ∧ is_add_route(c) ∧
                                        def_route(c) = r ∧ ports(c) = pts)),
   4 : p ∈ open_ports(r, t)    ⟺    (p ∈ associated_ports(r, t) ∧ is_open(p, t))
       where : t, t' : Time; c : Ctrl; r : Route; p : Port; pts : Ports
end DEFINING_ROUTE.
```

Figure 7: Action of control messages on routes

Finally we introduce an operation open_ports which yields the set of open ports among those which are associated to the route. This operation will be used for routing as described in Clause 7a.

3.6 Arrivals of Data Messages

Arrivals of data messages (see Figure 8) are specified similarly to arrivals of control messages. Each data message m is provided with the instant of its arrival to the TN (arrival(m)), the instant when it is completely received by the TN and the routing information is decoded (reception(m)), the data port on which it arrives (entry(m)) and the route (route(m)) it has to be routed to. The predicate

spec : DATA_ARRIVAL
  use : OPENING_PORT, ROUTE_INDEX
sort : Data
operations :
  arrival, reception : Data $\rightarrow$ Time
  entry : Data $\rightarrow$ Port
  route : Data $\rightarrow$ Route
predicates :
  is_entering : Data Port Time
axioms :
  1 : $\mathsf{arrival(m)} \leq \mathsf{reception(m)} \wedge \mathsf{reception(m)} < \mathsf{arrival(m)} + \mathsf{T}$,
  2 : $\mathsf{is\_entering(m,p,t)} \wedge \mathsf{is\_entering(m',p,t)} \Rightarrow \mathsf{m} = \mathsf{m'}$,
  3 : $\mathsf{arrival(m)} = \mathsf{t} \Rightarrow \mathsf{is\_open(entry(m),t)}$,
  4 : $\mathsf{is\_entering(m,p,t)} \Longleftrightarrow \mathsf{arrival(m)} \leq \mathsf{t} \wedge \mathsf{t} \leq \mathsf{reception(m)} \wedge \mathsf{entry(m)} = \mathsf{p}$,
  where : m, m′ : Data; p : Port; t : Time;
end DATA_ARRIVAL.

Figure 8: Arrival of data messages

is_entering defines the occupation of data ports-in. The axioms are analogous to those of the specification CTRL except Axiom 3 which, according to Clause 2c, states that a data message may not arrive to a closed data port.

3.7 Transit of Messages

Specification module TRANSIT (see Figures 9 and 10) describes different stages of lifecycle of messages inside the TN and their transmission outside of the TN. These stages are represented by the predicates introduced in this module. Our understanding of the informal specification leads us to consider the following stages of the lifecycle of data messages:

- Message m is been receiving by data port-in p (is_entering(m, p, _)).[3]

- At the reception instant reception(m) the TN detects whether it can be routed to a port of its route route(m). It consists of checking whether there is an open port associated with the route (open_ports(route(m), reception(m)) $\neq \emptyset$). If such a port exists, the message turns to the waiting state (is_waiting_inside(m, _)). It corresponds to the situation when the message is not "too old" or has not been yet detected as such.

- If no such port exists at the reception time, the message is put to the faulty collection (is_in_fc(m, reception(m))).

- A waiting message (is_waiting_inside(m, _)) which will not become "too old" before the end of its transmission outside (i.e. whose transit time through the TN will not exceed the constant T) is routed to the one among those data ports-out which have been associated with its route at its reception instant and were open at that instant. This is the beginning of the transmission on the data port-out (is_leaving(m, p, _)).

- A waiting message (is_waiting_inside(m, _)) is moved to the faulty collection (is_in_fc(m, _)) after being detected as "too old".

- From the faulty collection a message moves at some moment to the control port-out after after the reception of a send-faults control message. This is the beginning of the transmission on the control port-out (is_leaving_on_ctrl(m, _)).

- At some moment any transmission ends and the transmitted message is already outside the TN. Notice that this is represented by the fact that no atomic formula built from predicates is_entering, is_waiting_inside, is_in_fc, is_leaving or is_leaving_on_ctrl holds for the message.

A control message passes through the following states:

- Message c is been receiving by the control port-in (is_entering(c, _)).

- At the reception instant reception(c) the TN detects whether the message is incorrect (is_unrecognized(c)).

- After being interpreted, a correct control message disappears inside of the TN.

- An incorrect control message is put to the faulty collection at the reception time (is_in_fc(m, reception(m))).

- Form the faulty collection a control message moves as any data message at this stage.

spec : $\mathsf{TRANSIT}$
use : $\mathsf{DEFINING\_ROUTE, DATA\_ARRIVAL}$
predicates :
$\mathsf{is\_waiting\_inside, is\_in\_fc, is\_leaving\_on\_ctrl : Data\ Time}$
$\mathsf{is\_in\_fc, is\_leaving\_on\_ctrl : Ctrl\ Time}$
$\mathsf{is\_leaving : Data\ Port\ Time}$
axioms :

$$\begin{aligned}
1 &: \mathsf{is_waiting_inside}(m,t) \wedge \mathsf{is_waiting_inside}(m,t'') \wedge t < t'' \Rightarrow ((t \leq t' \wedge t' \leq t'') \Rightarrow \mathsf{is_waiting_inside}(m,t')),\\
2 &: \mathsf{is_in_fc}(m,t) \wedge \mathsf{is_in_fc}(m,t'') \wedge t < t'' \Rightarrow ((t \leq t' \wedge t' \leq t'') \Rightarrow \mathsf{is_in_fc}(m,t')),\\
3 &: \mathsf{is_in_fc}(c,t) \wedge \mathsf{is_in_fc}(c,t'') \wedge t < t'' \Rightarrow ((t \leq t' \wedge t' \leq t'') \Rightarrow \mathsf{is_in_fc}(c,t')),\\
4 &: \mathsf{is_leaving_on_ctrl}(m,t) \wedge \mathsf{is_leaving_on_ctrl}(m,t'') \wedge t < t'' \Rightarrow ((t \leq t' \wedge t' \leq t'') \Rightarrow \mathsf{is_leaving_on_ctrl}(m,t')),\\
5 &: \mathsf{is_leaving_on_ctrl}(c,t) \wedge \mathsf{is_leaving_on_ctrl}(c,t'') \wedge t < t'' \Rightarrow ((t \leq t' \wedge t' \leq t'') \Rightarrow \mathsf{is_leaving_on_ctrl}(c,t')),\\
6 &: \mathsf{is_leaving}(m,p,t) \wedge \mathsf{is_leaving}(m,p,t'') \wedge t < t'' \Rightarrow ((t \leq t' \wedge t' \leq t'') \Rightarrow \mathsf{is_leaving}(m,p,t')),\\
7 &: t < \mathsf{reception}(m) \Rightarrow (\neg \mathsf{is_waiting_inside}(m,t) \wedge \neg \mathsf{is_in_fc}(m,t)),\\
8 &: t < \mathsf{reception}(c) \Rightarrow \neg \mathsf{is_in_fc}(c,t),\\
9 &: \mathsf{is_waiting_inside}(m,t) \wedge \mathsf{is_in_fc}(m,t') \Rightarrow t < t',\\
10 &: \mathsf{is_in_fc}(m,t) \wedge \mathsf{is_leaving_on_ctrl}(m,t') \Rightarrow t < t',\\
11 &: \mathsf{is_in_fc}(c,t) \wedge \mathsf{is_leaving_on_ctrl}(c,t') \Rightarrow t < t',\\
12 &: \mathsf{is_waiting_inside}(m,t) \wedge \mathsf{is_leaving}(m,p,t') \Rightarrow t < t',
\end{aligned}$$

Figure 9: Internal flow and transmission of messages (part 1)

This is specified as follows

1. **Continuity and uniqueness**
   Any message can be in a given state only once. This means that, for a given message, predicates is_waiting_inside, is_in_fc, is_leaving and is_leaving_on_ctrl can hold on one time interval only. This is guaranteed by Axioms 1–6.

2. **Succession of stages: some necessary conditions**
   These are provided by Axioms 7–12 which describe minimal assumptions on the precedence of different stages of messages' lifecycles.

3. **Waiting state**
   Sufficient condition for reaching the waiting state is provided in Axiom 13.

4. **Transmission on data port-out**
   Axiom 14 is a necessary condition for a data message to be put on a data

[3] In this discussion we omit variables of sort Time which are replaced by an underscore.

13 : $\mathsf{open\_ports(route(m), reception(m))} \neq \emptyset \Rightarrow \mathsf{is\_waiting\_inside(m, reception(m))}$,
14 : $\mathsf{is\_leaving(m,p,t)} \Rightarrow \mathsf{p} \in \mathsf{open\_ports(route(m), reception(m))} \wedge \neg\mathsf{is\_in\_fc(m,t')}$),
15 : $(\mathsf{is\_waiting\_inside(m,t)} \wedge \forall \mathsf{t'}\ \neg\mathsf{is\_in\_fc(m,t')}) \Rightarrow \exists \mathsf{t''}\ \exists \mathsf{p}\ \mathsf{is\_leaving(m,p,t'')}$,
16 : $\mathsf{is\_leaving(m,p,t)} \Rightarrow \exists \mathsf{t'}\ (\mathsf{t} < \mathsf{t'} \wedge \neg\mathsf{is\_leaving(m,p,t')})$,
17 : $(\mathsf{open\_ports(route(m), reception(m))} \neq \emptyset \wedge \mathsf{is\_in\_fc(m,t)}) \Rightarrow \mathsf{arrival(m)} + \mathsf{T} < \mathsf{t}$,
18 : $\mathsf{open\_ports(route(m), reception(m))} = \emptyset \Rightarrow \mathsf{is\_in\_fc(m, reception(m))}$,
19 : $(\mathsf{is\_waiting\_inside(m,t)} \wedge \mathsf{arrival(m)} + \mathsf{T} < \mathsf{t}) \Rightarrow \exists \mathsf{t'}\ (\mathsf{t} < \mathsf{t'} \wedge \mathsf{is\_in\_fc(m,t')})$,
20 : $\mathsf{is\_leaving\_on\_ctrl(m,t)} \Rightarrow \exists \mathsf{c}\ (\mathsf{arrival(c)} < \mathsf{t} \wedge \mathsf{is\_in\_fc(m, reception(c))})$,
21 : $(\mathsf{is\_in\_fc(m, reception(c))} \wedge \mathsf{is\_send\_faults(c)}) \Rightarrow$
$\exists \mathsf{t}\ (\mathsf{reception(c)} < \mathsf{t} \wedge \mathsf{is\_leaving\_on\_ctrl(m,t)})$,
22 : $\mathsf{is\_leaving\_on\_ctrl(m,t)} \Rightarrow \exists \mathsf{t'}\ (\mathsf{t} < \mathsf{t'} \wedge \neg\mathsf{is\_leaving\_on\_ctrl(m,t')})$,
23 : $\mathsf{unrecognized(c)} \Longleftrightarrow \mathsf{is\_in\_fc(c, reception(c))}$,
24 : $\mathsf{is\_leaving\_on\_ctrl(c,t)} \Rightarrow \exists \mathsf{c'}\ (\mathsf{arrival(c')} < \mathsf{t} \wedge \mathsf{is\_in\_fc(c, reception(c'))})$,
25 : $(\mathsf{is\_in\_fc(c, reception(c'))} \wedge \mathsf{is\_send\_faults(c')}) \Rightarrow$
$\exists \mathsf{t}\ (\mathsf{reception(c')} < \mathsf{t} \wedge \mathsf{is\_leaving\_on\_ctrl(c,t)})$,
26 : $\mathsf{is\_leaving\_on\_ctrl(c,t)} \Rightarrow \exists \mathsf{t'}\ (\mathsf{t} < \mathsf{t'} \wedge \neg\mathsf{is\_leaving\_on\_ctrl(c,t')})$,
27 : $\mathsf{is\_leaving(m,p,t)} \wedge \mathsf{is\_leaving(m',p,t)} \Rightarrow \mathsf{m} = \mathsf{m'}$,
28 : $\mathsf{is\_leaving\_on\_ctrl(m,t)} \wedge \mathsf{is\_leaving\_on\_ctrl(m',t)} \Rightarrow \mathsf{m} = \mathsf{m'}$,
29 : $\mathsf{is\_leaving\_on\_ctrl(c,t)} \wedge \mathsf{is\_leaving\_on\_ctrl(c',t)} \Rightarrow \mathsf{c} = \mathsf{c'}$,
30 : $\neg(\mathsf{is\_leaving\_on\_ctrl(m,t)} \wedge \mathsf{is\_leaving\_on\_ctrl(c,t)})$,
31 : $\neg(\mathsf{is\_leaving\_on\_ctrl(m,t)} \wedge \exists \mathsf{p}\ \mathsf{is\_leaving(m,p,t')})$,
32 : $(\mathsf{is\_leaving(m,p,t)} \wedge \mathsf{is\_leaving(m,p',t')}) \Rightarrow \mathsf{p} = \mathsf{p'}$
where : t, t' : Time ; m, m' : Data ; c, c' : Ctrl; p, p': Port
end TRANSIT.

Figure 10: Internal flow and transmission of messages (part 2)

port-out. A sufficient condition is stated in Axiom 15. Axiom 16 tells us that a data message cannot occupy a data port-out for an infinite time.

5. **Data messages in the faulty collection**
   Axiom 17 is a necessary condition for a data message with a correct route to be put in the faulty collection. Sufficient conditions are stated in Axioms 18 and 19. Axiom 21 describes the situation when a data message leaves the faulty collection and enters the control port-out.

6. **Transmission of faulty data messages**
   A necessary condition is provided by Axiom 20 and the sufficient condition by Axiom 21. Axiom 22 tells us that a data message cannot occupy a control port-out during an infinite time.

7. **Control messages in the faulty collection**
   The necessary and sufficient condition for a control message to be put to the faulty collection is Axiom 23.

8. **Transmission of faulty control messages**
   This is described by Axioms 24, 25 and 26 which are analogous of Axioms 20 and 21 for faulty data messages.

9. **Mutual exclusion on ports-out**
   This is defined in Axioms 27–30.

10. Axioms 31 and 32 describe the fact the a data message can be sent only through one port-out. These are probably redundant with respect to the other axioms.

4 Conformity of the Formal Specification

In this section we address the problem of the conformity of our formal specification with respect to the informal requirements of the Transit Node system including modifications and assumptions listed in Section 2.1.

It is clear that, in general, one cannot know whether a formal specification truly describes the required system. One can only check that some essential properties of the system are satisfied by the theory described by the formal specification. This increases one's confidence in the conformity of the formal specification with respect to the informal requirements. In the case of algebraic specifications this amounts to proving that the properties we are interested in are the logical consequences of the axioms of the specification.

For the conformity of our TN specification we need to distinguish between two kinds of properties: *necessity properties* and *possibility properties*.[4] In order to make clear the distinction between them, note that any model of the specification represents a complete scenario of the functioning of the TN. This leads to the following remarks:

- The specification satisfies "P *is possible*" if and only if there is at least one model of the specification satisfying the property P. Consequently, "P *is possible*" is a consequence of the specification if and only if the specification augmented with P is consistent.

- The specification satisfies "P *is necessary*" if and only if P is a consequence (in the usual sense) of the specification.

Note that refutational theorem proving seems to be suitable for testing the validity of both kinds of properties.

Possibility properties will distinguished by the mention "**it is possible that**". For necessity properties we do not make a special mention. Some properties will

[4]This distinction corresponds with the usual modalities of temporal logics. It is not very surprising to rediscover them in the context of the temporal properties we deal with.

be shortened using the predicate is_inside : DataTime, which reflects the fact that a data message is inside the TN:

$$\text{is\_inside}(m,t) \iff (\text{is\_waiting\_inside}(m,t) \vee \exists p\ \text{is\_leaving}(m,p,t) \vee \text{is\_in\_fc}(m,t) \vee \text{is\_leaving\_on\_ctrl}(m,t))$$

Another useful predicate, already_sent : DataTime, reflects the fact that a data message has passed through the TN:

$$\text{already\_sent}(m,t) \iff (\text{reception}(m) < t \wedge \neg\text{is\_inside}(m,t))$$

We discuss below the conformity of our formal specification with each clause of the informal specification and we possibly state the corresponding property to be checked.

1. This clause is obviously satisfied since any model of PORT_INDEX (resp. ROUTE_INDEX) is a set of cardinality N (resp. M).

2a. According to Axioms CTRL(6), DATA_ARRIVAL(2), TRANSIT(27–30), two distinct messages cannot be on the same port at the same moment.

2b. Here, we want to check if arrivals and/or departures of messages may occur simultaneously. For instance, in order to know whether two data ports-in can simultaneously receive messages we check that **it is possible that**

$$\exists\, t\ \exists\, m\ \exists\, m'\ \exists\, p\ \exists\, p' \quad p \neq p' \wedge \text{is\_entering}(m,p,t) \wedge \text{is\_entering}(m',p',t)$$

2c. This clause is exactly expressed by Axiom DATA_ARRIVAL(3).

3. Even if the specification does not express any priority between ports or at the level of message transit, it is clear that fairness properties cannot be treated in the algebraic and classical logic frameworks.

4. We check the following property

$$\forall\, m\ \exists\, t \quad \text{arrival}(m) < t \Rightarrow (\neg(\exists p\ \text{is\_entering}(m,p,t) \vee \text{is\_inside}(m,t)) \vee \text{is\_in\_fc}(m,t))$$

5. The problem of the initial state has been deliberately left unspecified. We do not even specify that the TN has to have the beginning of its history. Moreover, our modification (see Section 2.1) overrides this clause. At the initial state, if there is one, all data ports are closed (see Figure 5), all routes are empty (see Axiom DEFINING_ROUTE(1)) and both control port-in and control port-out are open (and remain open).

6a. The reception of a control message c of type is_add_port makes the data port port(c) open (see Figure 5). We may additionally verify the property **it is possible that**

$$\forall\, p\ \forall\, t\ \exists\, t'\ \exists\, m \quad (\text{is\_open}(p,t) \wedge t \leq t') \Rightarrow \text{is\_entering}(p,m,t')$$

This would ensure that an open data port-in may receive messages.

6b. This clause corresponds to Axioms DEFINING_ROUTE(2, 3).

6c. This clause corresponds to Axioms TRANSIT(21, 25).

7a. Properties which guarantee that faulty messages are those described in the informal specification will be discussed in the sequel. We state below a property that ensures that a data message, that is never faulty during its history, leaves the TN through an open data port-out associated with the route of the message at the time of its reception:

$$(\forall t\ \neg \mathsf{is\_in\_fc}(m,t)) \Rightarrow$$
$$\exists t'\ \exists p\ (p \in \mathsf{open\_ports}(\mathsf{route}(m), \mathsf{reception}(m)) \wedge \mathsf{is\_leaving}(m,p,t'))$$

7b. Due the modifications (see Section 2.1) which overrides this clause, we read it: *no data message may disappear inside the TN.* This is stated as follows:

$$(\mathsf{reception}(m) \leq t \wedge (\exists t'\ ((\exists p\ \mathsf{is\_leaving}(m,p,t') \Rightarrow t \leq t') \vee$$
$$(\mathsf{is\_leaving\_on\_ctrl}(m,t') \Rightarrow t \leq t'))) \quad \Rightarrow \quad \mathsf{is\_inside}(m,t)$$

7c. We did not define a faulty state. According to Clause **9** faulty messages are those which go to the faulty collection. Consequently, this clause corresponds to Axiom TRANSIT(19).

8. The fact that ports-out transmit the message outside the TN corresponds to the formula

$$(\mathsf{is\_leaving}(m,p,t) \vee \mathsf{is\_leaving\_on\_ctrl}(m,t)) \Rightarrow \exists t'\ \mathsf{already\_sent}(m,t')$$

In order to show that messages can leave the TN in any order we may check that **it is possible that**

$$\exists m\ \exists m'\ (\mathsf{arrival}(m) \leq \mathsf{arrival}(m') \wedge \exists t\ (\mathsf{is\_inside}(m,t) \wedge \mathsf{already\_sent}(m',t)))$$

and that **it is possible that**

$$\exists m\ \exists m'\ (\mathsf{arrival}(m) \leq \mathsf{arrival}(m') \wedge \exists t\ (\mathsf{is\_inside}(m',t) \wedge \mathsf{already\_sent}(m,t)))$$

9. This clause corresponds Axioms TRANSIT(18, 19, 23).

10a. Faulty control messages are precisely those in the faulty collection. This clause corresponds therefore to Axiom TRANSIT(23).

10b&c. We did not define a s faulty state. According to Clause **9** faulty messages are those which go to the faulty collection. Consequently Axioms TRANSIT(17,18, 19) describe precisely properties required by Clause **10b&c**.

11a. Due to the latter remark this corresponds to Axiom TRANSIT(19).

11b. For this clause we may check that **it is possible that**

$$\exists m\ \exists t\ (\mathsf{is\_waiting\_inside}(m,t) \wedge \mathsf{arrival}(m) + T < t)$$

11c. This corresponds to Axioms TRANSIT(21, 25).

12. Nothing to check.

Of course the properties stated above should be proved. The use of a theorem prover would make easier this task.

5 Conclusions

We have presented an algebraic specification of the Transit Node system. The underlying formalism is Many-Sorted First Order Logic with Equality. Consequently we do not use techniques devoted to concurrency or real time. The idea which has allowed us to overcome the usual limitations of "classical" algebraic specifications to express concurrent and real time properties was the use of an abstract data type which make explicit temporal aspects in the system specification.

The example of the TN seems to be representative for a large class of concurrent and real time systems which can be characterized as communicating processes exchanging static data.[5] It shows that within classical many-sorted logic it is possible to specify concurrent and real time systems. Moreover this kind of specifications has the following advantages:

- Apart from modularity aspects, the semantics is simple and well known. It allow to express the true concurrency (no need of interleaving semantics).
- On the contrary to algebraic specifications of concurrency, the same formalism is used to express data type and static components properties of a system as well as its dynamic behaviour.
- Proof techniques which might be used to check the conformity of a formal specification with the informal requirements are well known and several tools which support such proofs are available or under development.

The limits of this approach are intrinsic to first order logic. It is mainly the matter of the incompleteness of theories which would be useful for modeling temporal aspects. We also notice that classical first order logic seems to be too fine for specifying simple dynamic aspects. For instance, in order to express that two actions a and a' (represented by two predicates depending on time) should succeed each other one needs to write:

$$(\mathsf{a}(\mathsf{t}) \wedge \mathsf{a}'(\mathsf{t}')) \Rightarrow (\mathsf{t} < \mathsf{t}' \wedge (\mathsf{t} < \mathsf{t}' \wedge \mathsf{t}' < \mathsf{t}'' \Rightarrow (\mathsf{a}(\mathsf{t}') \vee \mathsf{a}'(\mathsf{t}'))))$$
$$\mathsf{a}(\mathsf{t}) \Rightarrow \exists \mathsf{t}' \; \mathsf{a}'(\mathsf{t}')$$

This might be avoided in an approach which includes a logic of time intervals, for instance in the style of [4]. We believe that, more generally, the use of partial order builtin to the semantics would make temporal causalities between actions easier to express, especially in the contexts of recent results in resolution-style theorem proving. Among recent advances in this area we may cite [2] where authors provide a refutationally complete set of inference rules which besides the usual rules such that ordered resolution, ordered paramodulation or superposition include a new rule called *chaining* for dealing with transitive relations. Taking these results into account, our future work will include some experiments with the system "Saturate"

[5] This class does not include systems in which a process may be sent as a datum to another process.

[6] implementing those techniques. This tool is currently under development at Universitat Politècnica de Catalunya (Barcelona).

We did not discuss in this paper modularity issues. However, a careful look at the structure of our specification leads to remark that dependencies between modules reflect the temporal causality between different kinds of events. For instance, since no data may arrive to a closed port, module DATA_ARRIVAL uses module OPENING_PORTS which in turn uses module CTRL because control messages cause port opening.

Acknowledgments I gratefully acknowledge André Arnold, Michel Bidoit and David Janin for stimulating discussions. This work is partially supported by CNRS GDR de Programmation – Working Group VTT.

References

[1] E. Astesiano and G. Reggio. Algebraic Specification of Concurrency. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification*, LNCS 655, pages 1–39, Dourdan, Sept. 1991. Selected papers from the 8^{th} Workshop on Specification of Abstract Data Types.

[2] L. Bachmair and H. Ganzinger. Rewrite Techniques for Transitive Relations. Technical Report MPI–I–93–249, Max-Planck-Institut für Informatik, Saarbrücken, Nov. 1993.

[3] M. Bidoit. Pluss, un langage pour le développement de spécifications algébriques modulaires. Thèse d'Etat, Université de Paris-Sud, 1989.

[4] J. Y. Halpern and Y. Shoham. A Propositional Modal Logic of Time Intervals. *J. ACM*, 38(4):935–962, 1991.

[5] T. Knapik. *Spécifications algébriques observationnelles modulaires: une sémantique fondée sur une relation de satisfaction observationnelle.* PhD thesis, Université de Paris-Sud, Mar. 1993.

[6] P. Nivela and R. Nieuwenhuis. Saturation of First-order (Constrained) Clauses with the Saturate System. In C. Kirchner, editor, *Rewriting Techniques and Applications*, LNCS 690, pages 446–451, Montreal, June 1993.