Specification of the Transit Node in PSF\textsubscript{d}

S. Mauw F. Wiedijk

University of Amsterdam
Programming Research Group
P.O.Box 41882, 1009 DB Amsterdam
The Netherlands

abstract The specification language PSF\textsubscript{d} is used to give a formal specification of a transit node, a common case study in ESPRIT project METEOR. The design of the specification derived from the informal text and the ERAE specification is included. A short discussion on the relation to the specification in ERAE is provided.

1. INTRODUCTION

This paper contains a case study in the formal description technique PSF\textsubscript{d}. We specify a transit node, which is the common case study for several formalisms in the ESPRIT project METEOR. In [MHB89] the transit node is specified in the algebraic specification language PLUSS. The PSF\textsubscript{d} specification is derived partially from an informal text and partially from the ERAE specification in [Hag88]. The design of the specification is included, from which a general method can be derived for specifying similar problems in PSF\textsubscript{d}.

In [MHB89] the transit node is specified in the algebraic specification language PLUSS. The PSF\textsubscript{d} specification can be viewed as more implementation directed specification than the one in ERAE. Certain design decisions are made, e.g. in identifying the separate objects that act in parallel. Thus the PSF\textsubscript{d} specification, viewed as an implementation of the ERAE specification must be verified or validated. A short discussion is devoted to this topic.

2. PSF\textsubscript{d}

PSF\textsubscript{d} (Process Specification Formalism - Draft) is a Formal Description Technique developed for specifying concurrent systems. The formal definition of PSF\textsubscript{d} can be found in [MV88]. In [MV89] an introduction to the basic features is given.

PSF\textsubscript{d} has been designed as the base for a set of tools to support ACP (Algebra of Communicating Processes) [BK86]. We use bisimulation semantics to attach a meaning to the specification of processes. The part of PSF\textsubscript{d} that deals with the description of the data is based on ASF (Algebraic Specification Language) [BHK89]. Here we use initial algebra semantics.

PSF\textsubscript{d} supports the modular construction of specification and parameterization of modules.
3. THE TRANSIT NODE

The Transit Node is a case study, which was defined in the RACE project 1046 (SPECS). An informal description of the Transit Node and the ERAE specification of it can be found in [Hag88]. The informal specification reads as follows:

"The system to be specified consists of a transit node with:
- 1 Control Port-In
- 1 Control Port-Out
- \(N\) Data Ports-In
- \(N\) Data Ports-Out
- \(M\) Routes Through

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

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

The node is "fair". All messages are equally likely to be treated, when a selection must be made, and all messages will eventually transit the node, or be placed in the collection of faulty messages.

Initial State: 1 Control Port-In, 1 Control Port-Out.

The Control Port-In accepts and treats the following three messages:
- \(\text{Add-Data-Port-In} & \text{-Out}(n)\)
  gives the node knowledge of a new port-in\((n)\) and a new port-out\((n)\). The node commences to accept and treat messages sent to the port-in, as indicated below on Data Port-In.
- \(\text{Add-Route}(m), n(i), n(j), ..., l)\)
  gives the node knowledge of a route associating route \(m\) with Data Port-Out\(n(i), n(j), ..., l)\).
- \(\text{Send-Faults}\)
  routes all saved faulty messages, if any to Control-Port-Out. The order in which the faulty messages are transmitted is not specified.

A Data Port-In accepts and treats only messages of the type:
- \(\text{Route}(m), \text{Data}\)
  The Port-In routes the message, unchanged, to any one (non-determinate) of the Data Ports-Out associated with route \(m\). (Note that a Data Port-Out is serialized - the message has to be buffered until the Data Port-Out can process it). 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 \(T\).

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.

All faulty messages are saved until a Send-Faults command message causes them to be routed to Control Port-Out. Faulty messages are messages on the Control Port-In that are not one of the three commands listed, messages on a Data Port-In that indicate an unknown route, or messages whose transit time through the node is greater than \(T\). Messages that exceed the transit time of \(T\) become faulty as soon as the time \(T\) is exceeded.

It is permissible for a faulty message to not be routed to Control Port-Out (because, for example, it has just become faulty, but has not yet been placed in a faulty message collection), but all faulty messages must eventually be sent to Control Port-Out with a succession of Send-Faults commands.

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."
4. DESIGN OF THE SPECIFICATION

4.1. General

The specification was designed using a mixed top-down and bottom-up approach. It was based on the informal text, while using the interpretation of the text in the ERAE specification when needed to fill in omissions or solve ambiguities. Several design decisions were made, which did not follow directly from the informal description of the case study. (e.g. the decision to let the Control Port-in keep control of the table containing all routes through the node).

4.2. Design

We first identify all parameters of the system, i.e. objects which are - and should be- unspecified. Since "it may be assumed that a source of time is available in the environment", we postulate the existence of a process that behaves like a clock. This can be done by making a parameter containing this clock process. The second parameter is formed by the time that a message may be inside the node without getting faulty, the maximal transit time. The exact length of this duration should be decided upon at the implementation phase.

Then we identify all (concurrent) components in the system. We have a Control-Port-In, a Control-Port-Out, a number of Data-Ports-in and a number of Data-Ports-Out. Note that we don’t consider the Routes as components, since these are static objects without temporal behaviour. Because all Data-Ports-In have the same behaviour, we can specify just one process, indexed with the actual name of the port. The same holds for the Data-Ports Out.

Now we make the decision that the routes and the information about the ports that exist are handled by the Control-Port-In, so this process is indexed with a route-table and with a port-set. Furthermore we see that the Control-Port-Out must contain a number of faulty messages that should be flushed and that every Data-Port-Out must contain a number of messages that should be sent to the environment. So both processes are indexed with a message-bag. The signature of the top-level objects now looks like:

```
processes
    control-port-in : route-table # port-set
    control-port-out : message-bag
    data-port-in : port-name
    data-port-out : port-name # message-bag
```

From the informal text and the ERAE specification we can now define the initial state of the the node. It consists of the concurrent operation of the control-port-in and the control-port-out, indexed with the empty-route-table, the empty-port-set and the empty-message-bag. Of course we must add the parameter process clock in parallel and we must abstract from the internal actions and encapsulate unsuccessful communications.

```
    transit-node = hide(I, encaps(H,
        clock ||
        control-port-in(empty-route-table, empty-port-set) ||
        control-port-out(empty-message-bag)))
```

Now we can proceed in a bottom up way by defining the data types route-table (an instance of the parameterized module table with the data type routes), port-set (sets instantiated with ports), message-bag (bags instantiated with messages) and port-name.
The top-down approach is continued by defining the behaviour of the four processes, each in a separate module. This leads to the question which objects are connected, in order to communicate to each other. We see that there is a link between the control-port-in and the control-port-out. Every data-port-in is linked to the control-port-in for route information and to the control-port-out for sending faulty messages. All data-ports-in are connected to all data-ports-out to transmit messages. And finally all ports have a connection to the environment for either accepting or transmitting messages.

As can be seen in the specification, the behaviour of the objects is specified by determining all initial communication actions. Every action is then followed by the corresponding behaviour, e.g. a transmission or a state change. This can possibly be specified by using subprocesses.

The control-port-in e.g. can accept one of the following messages:

- add-datum-port(p), followed by the subprocess that handles adding a data-port-in and a data-port-out;
- add-route(r), followed by a state change where the route-table is updated;
- send-faults, followed by forwarding this message to control-port-out;
- request-route(rn), followed by sending appropriate information about the route back.

After having identified all atomic actions (i.e. communication attempts) we can define the communication function and the set of atoms that has to be encapsulated and abstracted.

4.3. Topology of the transit node

We can visualize the structure of the transit node with the following picture.
5. THE SPECIFICATION

The specification that resulted from the design as described in the previous paragraph will now be given. Note that the linear structure of the specification does not comply with the way the specification was designed. This is because the formalism forces us to write down the specification in a bottom-up way.

We first give all basic data types needed in the specification, then we define the data types specific to the transit node, then we define all processes involved and finally we give an example of an instantiation of the clock parameter.

5.1. Basic data types

The basic data types consist of the simple types booleans and natural numbers, and the parameterized types bags, sets and tables. The difference between bags and sets is that in a set duplicates are removed. A table can be used to look up an item corresponding to the value of a certain key.

```plaintext
data module booleans begin
  exports begin
    sorts BOOL
    functions
      true : -> BOOL
      false : -> BOOL
      or : BOOL # BOOL -> BOOL
      and : BOOL # BOOL -> BOOL
  end
  variables
    b : -> BOOL
  equations
    [1] or(true, b) = true
    [2] or(false, b) = b
    [3] and(true, b) = b
    [4] and(false, b) = false
end booleans
```

```plaintext
data module natural-numbers begin
  exports begin
    sorts nat
    functions
      0 : -> nat
      s : nat -> nat
      eq : nat # nat -> BOOL
      lt : nat # nat -> BOOL
      + : nat # nat -> nat
      - : nat # nat -> nat
  end
  imports booleans
```
variables
  n, n1, n2 : -> nat

equations
  [1] eq(0, 0)         = true
  [2] eq(0, s(n))     = false
  [3] eq(s(n), 0)     = false
  [4] eq(s(n1), s(n2)) = eq(n1, n2)
  [5] lt(0, s(n))     = true
  [6] lt(n, 0)        = false
  [7] lt(s(n1), s(n2)) = lt(n1, n2)
  [8] n + 0          = n
  [9] n1 + s(n2)      = s(n1 + n2)
 [10] 0 - n          = 0
 [11] n - 0          = n
 [12] s(n1) - s(n2)  = n1 - n2

end natural-numbers

data module bags
begin

  parameters
  items
    begin
    sorts item
    end items

  exports
  begin
    sorts bag
    functions
      empty-bag : -> bag
      add : item # bag -> bag
    end

  variables
    i1, i2 : -> item
    b       : -> bag

  equations
    [1] add(i1, add(i2, b)) = add(i2, add(i1, b))

end bags

data module set
begin

  parameters
    equality
      begin
        functions
          eq : item # item -> BOOL
        end equality

  exports
  begin
    functions
      eq   : set # set -> BOOL
      element : item # set -> BOOL
    end

end
imports
  bags
  { renamed by
    [ bag -> set,
      empty-bag -> empty-set]
  },
  booleans
variables
  i, i1, i2 : -> item
  s : -> set
equations
  [1] add(i, add(i, s)) = add(i, s)
  [2] element(i, empty-set) = false
  [3] element(i1, add(i2, s)) = or(eq(i1, i2), element(i1, s))
end set

data module tables
begin
parameters
  items
  begin
    sorts key, value
    functions
      eq : key # key -> BOOL
      default-value : -> value
    end items

exports
  begin
    sorts table
    functions
      empty-table : -> table
      add : key # value # table -> table
      look-up : key # table -> value
    end

imports booleans

variables
  k, k1, k2 : -> key
  v : -> value
  t : -> table
equations
  [1] look-up(k, empty-table) = default-value
  [2] look-up(k1, add(k2, v, t)) = if(eq(k1, k2), v, look-up(k1, t))
end tables
5.2. Data types specific to the transit node

The module \texttt{time} supplies functions to deal with timing information. To the outside the sort \texttt{time} is built up from the constant \texttt{initial-time}, using the \texttt{+}-function to add durations. A \texttt{duration} is either the constant \texttt{tick-duration}, or the difference of two times. Internally we use the \texttt{naturals} and auxiliary functions to define the exported functions.

\begin{verbatim}
data module time begin
  exports
  begin
    sorts time, duration
    functions
    initial-time : -> time
    tick-duration : -> duration
    lt : duration # duration -> BOOL
    _ + _ : time # duration -> time
    _ - _ : time # time -> duration
  end

  imports natural-numbers

  functions
  time : nat -> time
  duration : nat -> duration

  variables
  n1, n2 : -> nat

  equations
  [1] initial-time = time(0)
  [2] tick-duration = duration(s(0))
  [3] lt(duration(n1), duration(n2)) = lt(n1, n2)
  [4] time(n1) + duration(n2) = time(n1 + n2)
  [5] time(n1) - time(n2) = duration(n1 - n2)

end time
\end{verbatim}

The type of information that can be transmitted through the transit node is defined in the module \texttt{datum}.

\begin{verbatim}
data module datum begin
  exports
  begin
    sorts datum
  end

  imports natural-numbers

  functions
  datum : nat -> datum

end datum
\end{verbatim}
The transit nodes contain a number of ports for input and output. These ports are named with natural numbers. Port names can be collected into sets by binding the parameter of the basic module set to port-name.

```plaintext
data module port-name
begin
  exports
  begin
    sorts
    port-name
    functions
    eq : port-name # port-name -> BOOL
  end

  imports natural-numbers
  functions
  port-name : nat -> port-name

  variables
  n1, n2 : nat

  equations
  {1} eq(port-name(n1), port-name(n2)) = eq(n1, n2)

end port-name
```

data module port-sets
begin

  imports
  set
  { renamed by
    [ set -> port-set,
      empty-set -> empty-port-set ]
  } items bound by
  [ item -> port-name ]
  to port-name
  equality bound by
  [ eq -> eq ]
  to port-name

end port-sets

A route consists of a route-name and a set of output ports associated with this route. Routes are collected into tables in order to look up the port-set corresponding to the name of a previously created route.

```plaintext
data module route-names
begin

  exports
  begin
    sorts
    route-name
    functions
    eq : route-name # route-name -> BOOL
  end
```

imports natural-numbers
functions
  route-name : nat -> route-name

variables
  n1, n2 : -> nat

equations
  [1] eq(route-name(n1), route-name(n2)) = eq(n1, n2)

end route-names

data module routes
begin
exports
begin
sorts route
functions
  route : route-name # port-set -> route
  name-of : route -> route-name
  ports-of : route -> port-set
  eq : route # route -> BOOL
end

imports boolean, port-sets, route-names
variables
  n1, n2 : -> route-name
  ps1, ps2 : -> port-set

equations
  [1] name-of(route(n1, ps1)) = n1
  [2] ports-of(route(n1, ps1)) = ps1
  [3] eq(route(n1, ps1), route(n2, ps2)) = and(eq(n1, n2), eq(ps1, ps2))

end routes

data module route-tables
begin
imports
  tables
  {renamed by
    [ table -> route-table,
      empty-table -> empty-route-table]
  items bound by
    [ key -> route-name,
      value -> port-set,
      eq -> eq,
      default-value -> empty-port-set]
  to routes}

end route-tables
If components communicate to the outside world or to each other, messages are exchanged. Most of the messages are indexed with a value of some data type. Messages can be collected in bags.

```
data module messages
begin

exports
begin
sorts message
functions
  add-datum-port : port-name -> message
  add-route : route -> message
  send-faults : -> message
  routed-datum : route-name # datum -> message
  req-route : route-name -> message
  available-ports : port-set -> message
  timed-message : time # datum -> message
  datum : datum -> message
end

imports datum, time, port-name, routes
end messages
```

data module message-bags
begin
imports

bags
renamed by

  { bag -> message-bag,
    empty-bag -> empty-message-bag }

items bound by

  [ item -> message ]

to messages

end message-bags

The various components of the transit node are connected to each other with channels. There are also channels to the environment.

```
data module channels
begin

exports
begin
sorts channel
functions
  control-input : -> channel
  control-output : -> channel
  control-in-to-out : -> channel
  control-to-data : port-name -> channel
  data-to-control : port-name -> channel
  rejection : -> channel
  data-in-to-out : port-name # port-name -> channel
  data-input : port-name -> channel
  data-output : port-name -> channel
end

imports port-name
end channels
```
5.3. The processes

5.3.1. Communication  The module \textit{communication} defines the atomic actions that can be executed by the various components, when trying to communicate. The communication function is defined such that a read action (\textit{r}) and a send action (\textit{s}) can be combined into a communication action (\textit{c}). These actions are indexed with the channel used to communicate and the message to be transmitted. In the same way timing information can be communicated.

The set of internal actions (\textit{I}) and the set of actions to be encapsulated in order to get only successful communication (\textit{H}) are also defined.

\begin{verbatim}
process module communication
begin

exports begin
atoms
  r : channel # message
  s : channel # message
  c : channel # message
  read-time : time
  send-time : time
  comm-time : time

sets of atoms
  I = { c(c, m), comm-time(t) |
        t in time, c in internal-channels, m in message }
  H = { r(c, m), s(c, m), send-time(t), read-time(t) |
        t in time, c in internal-channels, m in message }

end

imports
  channels,
  messages,
  time

sets of channel
  internal-channels =
    \{ control-in-to-out, rejection, data-to-control(pn1), control-to-data(pn1),
      data-in-to-out(pn1, pn2) | pn1 in port-name, pn2 in port-name \}

communications
  r(c, m) | s(c, m) = c(c, m)
  for c in channel, m in message
  read-time(t) | send-time(t) = comm-time(t)
  for t in time

end communication
\end{verbatim}
5.3.2. Data-ports-in For every port-name a process data-port-in is defined. Every data-port-in behaves as follows. First it reads from its input channel the message to send some datum along some route. Then it reads the current time and asks the control-port-in for the port set attached to the requested route. Then a transit attempt is made. If the route-name was faulty, an empty-port-set was returned and the incoming message is routed to the rejection channel, thus becoming faulty. If the port-set was not empty, one port is selected randomly and after adding a time stamp the incoming message is routed to that port. The process transit-datum is not defined in case the port-set is empty. This means that it equals deadlock.

```plaintext
process module data-ports-in
begin
  exports
  begin
    processes
      data-port-in : port-name
    end
  imports
    port-sets,
    route-names,
    time,
    communication
    processes
      transit-attempt : port-set # port-name # time # route-name # datum
      transit-datum : port-set # port-name # time # datum
  variables
    t1, t2 : -> time
    p1, p2 : -> port-name
    rn : -> route-name
    ps : -> port-set
    d : -> datum
  definitions
    data-port-in(p1) = sum(d in datum, sum(rn in route-name,
      r(data-input(p1), routed-datum(rn, d)))
    sum(t1 in time, read-time(t1) . s(data-to-control(p1), req-route(rn))
    sum(ps in port-set, r(control-to-data(p1), available-ports(ps))
      transit-attempt(ps, p1, t1, rn, d)
    data-port-in(p1)))
    transit-attempt(empty-port-set, p1, t1, rn, d) =
    s(rejection, routed-datum(rn, d))
    transit-attempt(add(p2, ps), p1, t1, rn, d) =
    transit-datum(add(p2, ps), p1, t1, d)
    transit-datum(add(p2, ps), p1, t1, d) =
    s(data-in-to-out(p1, p2), timed-message(t1, d)) +
    transit-datum(ps, p1, t1, d)
end data-ports-in
```
5.3.3. Data-ports-out The following module is parameterized with a duration, max-transit-time, that
determines the maximum time a message may stay within the transit node.
For every port-name a process data-port-out is defined. Every data-port-out is indexed with a bag of
messages that must be sent to the environment. Initially this bag is empty. It starts by reading a
timed message from one of the data-input-ports. This message is added to the bag and the process
starts again. If the bag is not empty, the process also has the possibility to output some message from
the bag. If the max-transit-time is expired, then the message becomes faulty and will be sent to the
rejection channel. Otherwise, the message is sent to the environment.

```plaintext
process module data-ports-out
begin

parameters
  max-transit-time
begin
functions
  max-transit-time : -> duration
end max-transit-time

exports
begin
  processes
  data-port-out : port-name # message-bag
end

imports
  port-name,
  message-bags,
  communication

processes
  handle-message-out : BOOL # datum # port-name

variables
  t, t1, t2 : -> time
  pl, p2 : -> port-name
  mb : -> message-bag
  d, e : -> datum

definitions
  data-port-out(p2, empty-message-bag) =
    sum(pl in port-name, sum(t1 in time, sum(d in datum,
      r(data-in-to-out(pl, p2), timed-message(t1, d)) .
    data-port-out(p2, add(timed-message(t1, d), empty-message-bag))))
  data-port-out(p2, add(timed-message(t2, e), mb)) =
    sum(pl in port-name, sum(t1 in time, sum(d in datum,
      r(data-in-to-out(pl, p2), timed-message(t1, d)) .
    data-port-out(p2,
      add(timed-message(t1, d), add(timed-message(t2, e), mb)))))) +
    sum(t in time, read-time(t) .
    handle-message-out(lt(t - t2, max-transit-time), e, p2) .
    data-port-out(p2, mb))

  handle-message-out(false, d, p2) =
    s(rejection, datum(d))
  handle-message-out(true, d, p2) =
    s(data-output(p2), datum(d))
end data-ports-out
```
5.3.4. Control-port-in The process control-port-in keeps track of all defined routes and all existing ports, so it is indexed with a route-table and a port-set. It is connected to the environment with the control-input channel. Via this channel it can receive the message to add a datum-port, to add a route, or to flush all faulty messages. As a last option it can receive a request from some data-port-in to send the routing information belonging to some route-name. All these incoming messages are treated separately. The request to add a datum port is handled using a subprocess. This handler checks whether the data port already exists. Then it either rejects the message or adds the port to the port-set and creates two new parallel processes: a data-port-in and a data-port-out.

If a request is made to add a route, it simply adds the route information to the route-set. A send-faults request is simply passed on to the control-port-out. A request for route information is answered by looking up the requested information and sending it back.

```
process module control-port-in
begin

exports
begin
processes
control-port-in : route-table # port-set
end

imports
route-tables,
communication,
data-ports-in,
data-ports-out

processes
handle-add-port : route-table # port-set # port-name # BOOL

variables
p : -> port-name
rt : -> route-table
ps : -> port-set

definitions
control-port-in(rt, ps) =
sum(p in port-name, r(control-input, add-datum-port(p))) .
handle-add-port(rt, ps, p, element(p, ps))
+ sum(r in route, r(control-input, add-route(r))) .
control-port-in(add(name-of(r), ports-of(r), rt), ps)
+ r(control-input, send-faults) .
s(control-in-to-out, send-faults) .
control-port-in(rt, ps)
+ sum(p in port-name, sum(rn in route-name,
r(data-to-control(p), req-route(rn)) .
s(control-to-data(p), available-ports(look-up(rn, rt)))))) .
control-port-in(rt, ps)
handle-add-port(rt, ps, p, true) =
s(rejection, add-datum-port(p)) .
control-port-in(rt, ps)
handle-add-port(rt, ps, p, false) =
control-port-in(rt, add(p, ps)) ||
data-port-in(p) || data-port-out(p, empty-message-bag)

end control-port-in
```
5.3.5. Control-port-out The process control-port-out is indexed with the message-bag containing all faulty messages. It has a simple behaviour. It can receive the message to send all faulty messages to the environment, which is handled by the subprocess flush, or it can receive faulty message via the rejection channel.

```haskell
process module control-port-out
begin

exports
begin
processes
  control-port-out : message-bag
end

imports
  message-bags,
  communication

processes
  flush : message-bag

variables
  m : -> message
  mb : -> message-bag

definitions
  control-port-out(mb) =
    r(control-in-to-out, send-faults) . flush(mb)
    + sum(m in message, r(rejection, m) .
        control-port-out(add(m, mb)))

  flush(empty-message-bag) = control-port-out(empty-message-bag)
  flush(add(m, mb)) = s(control-output, m) . flush(mb)

end control-port-out
```

5.3.6. Transit-node Finally the transit node is specified by the concurrent operation of the clock process, which is a parameter of the system, the control-port-in and the control-port-out. These ports are initialized with an empty table, set and bag. In order to hide internal actions and to get only successful communication, we add the hiding operator and the encapsulation operator. Note that apart from the parameter clock, we also inherit the parameter max-transit-time from the imported module data-ports-out.

```haskell
process module transit-node
begin

parameters
  time
  begin
    processes
      clock
    end
```
exports
begin
  processes
  transit-node
end

imports
  control-port-in,
  control-port-out

definitions
  transit-node = hide(I, encaps(H, clock ||
                              control-port-in(empty-route-table, empty-port-set) ||
                              control-port-out(empty-message-bag)))
end transit-node

5.4. Example of a clock

In this section we give an example of how the clock parameter of the transit node can be initialized. The process clock starts at the initial-time. Then it can do a tick, followed by an increment of the current time with a tick-duration, or it can send the time to anyone willing to read it. Note that in this version of a clock the action of sending the time will not cost any time.

process module a-clock
begin
  exports
    begin
      processes
        clock
    end

  imports
    time,
    communication

  atoms
    tick

  processes
    clock : time

  variables
    t : => time

  definitions
    clock = clock(initial-time)
    clock(t) = tick . clock(t + tick-duration) +
                 send-time(t) . clock(t)
end a-clock
process module transit-node-with-a-clock
begin

imports
  transit-node
  {time bound by
   [clock -> clock]
   to a-clock}

end transit-node-with-a-clock

5.5. Graphical representation of the import relation

Using the IDEAS tool developed within the METEOR project [Ide88] we can give the following picture (see figure 2), representing the import relation between all modules of the specification of the transit node. Rectangular boxes are used for data modules and boxes with rounded corners are used for process modules. An arrow from a module to another module means that the former is imported into the latter. Note that not all textual imports are present in the picture. We used a tool to compute the minimal import relation having the same transitive closure as the textual one.

6. RELATION TO THE ERASE SPECIFICATION

In this section we will give a brief discussion of the relation between the ERASE specification and the PSF_d specification of the transit node. It is clear that, since ERASE was designed for requirements specification, the first one is closer to the textual specification, whereas in the second one some design decisions had to be made. As an example look at the routing information that is treated as a separate entity in ERASE, while in PSF_d it is part of the state of the control-port-in.

The ERASE language is based on temporal logic. Its formal semantics can be found in [HR89], and [DHR88] contains an introduction to the use of ERASE.

In order to validate that a PSF_d specification is correct with respect to an ERASE specification, a formal treatment of this notion of validation would be needed. Since this paper does not focus on this subject, we only give some informal reasoning about the relation between the two specifications.

The validation is made up of two parts. First we must give a relation between the entities declared in the ERASE specification and the ones declared in the PSF_d specification, and then we must provide an interpretation of the temporal statements in ERASE into PSF_d.

6.1. Entities

A quick inspection learns that, apart from some design decisions and detail implementations, the entities in ERASE relate to the entities in PSF_d having the same name. So where ERASE contains messages like Add-route msgs indexed with a route nr and a series of out port-nr, PSF_d has a data type messages, containing a function add-route, indexed with route which is a combination of a route-name and a port-set.

As an other example look at the entity Data port-in which is indexed with a nr, and is able to receive Data msgs via a port. In PSF_d this translates to a process data-port-in, indexed with a port-name, having a channel to the environment called data-input, via which it can receive a routed-datum.
figure 2 The import relation
6.2. Temporal statements

Naively speaking the interpretation of a temporal statement in ERAE into PSFd consists of an interpretation of all events involved into atomic actions, followed by a verification that every possible trace of the specification in PSFd satisfies all temporal statements about events given in the ERAE specification. Unfortunately this approach is too simple since not only temporal information is involved but also information about the state space of the system.

As an example of how to informally validate the PSFd specification, we will give some ERAE statements and their informal interpretation in the PSFd specification.

\[
\text{initially } \Rightarrow \neg \exists \dpi: \text{is-in}(\dpi, \text{Data-ports-in}) \\
\quad \land \neg \exists \dpo: \text{is-in}(\dpo, \text{Data-ports-out}) \\
\quad \land \neg \exists \ri: \text{is-in}(\ri, \text{Routes}) \\
\quad \land \neg \exists \wm, \dm: \text{faulty}(\wm) \lor \text{faulty}(\dm)
\]

This can be interpreted as the statement that there are no data ports in the definition of the process \textit{transit-node}, and that the port-set, route-table and (faulty) message-bag are empty:

\[
\text{transit-node } = \text{hide}(I, \text{encaps}(E, \\
\text{clock } || \\
\text{control-port-in}(\text{empty-route-table}, \text{empty-port-set}) || \\
\text{control-port-out}(\text{empty-message-bag})))
\]

A number of statements are about the behaviour of the environment of the transit node. These statements are not explicitly met by the PSFd specification, since it only specifies the behaviour of the transit node without restricting its environment. As an example look at the statement

\[
\text{occurs}(\dm) \Rightarrow \bullet \text{ exists}(\text{port}(\dm))
\]

which states that messages only arrive at existing input ports (the symbol \(\bullet\) means "true in the previous state"). This assumption about the environment is not stated in the PSFd specification.

As a last example look at the statement about state changes concerning data-ports-in:

\[
\text{exists}(\dpi) \land \neg \text{ exists}(\dpi) \\
\Rightarrow \exists \ampm: \text{occurs}(\ampm) \land \text{nr}(\dpi) = \text{port-nr}(\ampm)
\]

This states that if a data-port-in is created, an add-port-message must have been occurred. In the PSFd specification this is verified by looking at all places where a data-port-in is created. This can only happen in the subprocess \textit{handle-add-port} of the process control-port-in. This subprocess is only invoked after the atomic action \(c(\text{control-input, add-datum-port}(p))\) has occurred for some appropriate \(\text{port-name} \ p\).

It is clear that this reasoning is very informal. This is because the existence of a data-port-in is easy to check at the textual level of the specification, but not at the level of the semantics of PSFd. The semantics is a labeled transition graph, which in no way contains information about the number of processes that it is constructed from, but only about the actions that can be performed by the system. Also the actual value of the indexes of the processes involved is not part of the semantics.
7. DISCUSSION

Since some design decisions were needed, the specification of the transit node in PSF_d is more specific than the specification in ERAE. There is no easy transformation from an ERAE specification to a PSF_d specification, however when having an ERAE specification, the informal text can be interpreted more easily.

We can only give an informal validation of the PSF_d specification when relating it to the ERAE specification. This is due to the fact that in some cases ERAE statements relate to the state of the system, which is not part of the formal semantics of PSF_d. We can however look at the textual level of the specification and give an informal reasoning. Also restrictions to the environment can not be expressed in PSF_d.

The design of the specification can be generalized to the following method:

- Identify the parameters of the system.
- Identify all concurrent components.
- Add indexes to the process names of each component to keep track of state information and to create more instances of the object.
- Define the abstract data types needed for these indexes.
- Specify how the components are connected.
- Define the initial state of the system.
- Define the behaviour of each component.

Of course the last step of this method can be very involved. Each component in turn can then be divided into subcomponents, in such a way that the method recursively applies to these subcomponents.

As a conclusion we can state that PSF_d is well suited for the specification of concurrent systems.

8. ACKNOWLEDGEMENTS

We like to thank Jos Baeten and Henrik Jacobsson for proof reading this document and the specification and Hans Mulder for technical advice.

9. REFERENCES


