Generation of Animations for Simulation of Process Algebra Specifications

Bob Diertens
University of Amsterdam
Department of Computer Science
Programming Research Group

Generation of Animations for Simulation of Process Algebra Specifications

Bob Diertens

Report P0003  
october 2000
Generation of animations for simulation of process algebra specifications

Bob Diertens

University of Amsterdam
Programming Research Group
e-mail: bobd@science.uva.nl

ABSTRACT

We present a tool for generation of animations from process algebra specifications for use in simulation. These animations can give a clear view of the simulation, and so they make testing easier.

The implementation of the tool is explained with the use of a few examples. Adaptations that have to be made to the specifications and some restrictions that apply are also explained.

1. Introduction

In [Die97] a platform is presented for simulation and animation of process algebra specifications. These animations have to be created by hand. So whenever the specification changes, the animation has to be adapted. This makes it difficult to use it for testing, especially for larger specifications.

We try to overcome this problem by generating animations from the specifications. At first, this seems an impossible job, because we examine processes statically which leaves us with open terms. But we want to find out to what extent we can do this and how we have to adapt the specifications in order to get better results.

We talk about specifications in PSF [MauVel90] [Die94] [DiePon94] that are compiled into TIL-code. The PSF-Toolkit\(^1\) [MauVel93] contains a compiler that translates PSF-code into TIL-code. But every process algebra specification language that can be compiled into TIL-code can be used.

2. Generating Animations

The animations we use consist of a description of a picture, an action-function that defines for each atom or communication what changes in the picture have to be made, and a choose-function that defines for each atom or communication to which choose-list from a process it must be added.

We have divided the problem of generating an animation from a specification in several steps. First, we have to analyze the specification with as result a process graph and a list of possible atoms and communications that can take place.

Secondly, we have to convert the process graph into a picture. We use the program `dot`,\(^2\) which calculates

---

1. The PSF-Toolkit is available at http://www.science.uva.nl/~psf/.
2. Dot is part of the software package Graphviz from AT&T Bell Laboratories.
coordinates for nodes and edges of a graph. We generate an animation from the output of *dot* by a Perl
[WalChrSch96] script.
Thirdly, we generate an action-function and a choose-function from the list of atoms and communications.
In section 2.1 we explain the implementation of the various parts and in section 2.2 we deal with several
difficulties in constructing a process graph.

2.1 Implementation
We explain the implementation of the various parts in general and use a specification of the
Alternating Bit Protocol in PSF (see A.1) as an example.

2.1.1 Process Graphs
We describe here the steps that we make in order to generate a graph from a specification. Several
steps could have been incorporated, but we have chosen to keep our code as simple as possible.

**Build Process Tree**
Fupor each definition of a process we build a process tree in which the nodes represent the operators and
processes, and the edges represent a list of atoms. These lists of atoms eliminate the sequential
operator (.).

**Expand Tree**
We take the process tree for the top process and expand it, by replacing the processes with their
process tree. This is done recursively, but a process is only expanded once in a tree since we already
have all possibilities. Except for the subtrees of a parallel operator (||), in which a process may be
expanded in each subtree, so that possible communications can be found.

**Mark up Tree**

*Put ID on Processes*
Give the top process of the tree and of the subtrees of a node that represents a parallel operator, an
ID. Mark all atoms with the process-ID of the subtree it belongs to.

*Fupind Sum Atoms*
We mark all atoms that can act as a sum-port. These are the atoms first in the list of atoms
belonging to the edge from the node for the sum operator to its subtree, and that have the variable
of this sum operator in one of their arguments.
This information is later used in deciding the type of a communication.

*Encapsulate and Hide Atoms*
We also mark the atoms that will be encapsulated or hidden. This information will be used later
in calculation of the communications. (We are matching open terms, so this can result in not
detecting an atom as a member of a set.)

*Fupind Communications*
We go down in the tree to the leaf nodes. From there we go up and list the atoms we encounter. When
we meet an encapsulation operator, we delete the atoms from our list that are encapsulated by this
operator. When we meet a hide operator we mark the atoms that are hidden by this operator. When
we meet a parallel operator, we calculate the possible communications between the atoms from the list
for each subtree, and assign this list of communications to this node.
Back at the top, we have collected a list of all atoms that can be performed.
Encapsulate and Hide Communications
We mark the communications that will be encapsulated or hidden.

Collect the communications
We go down in the tree and on our way up we list the communications. When we meet an encapsulation operator, we remove the communications that are encapsulated by this operator from the list.
At the top, we have collected a list of all possible communications.

Properties of the Processes
By inspecting the list of atoms and communications, we can see which of the processes are used. There is no need to put processes in the graph that are not used. However, for debugging purposes this is made optional.
We consider processes which contain atoms that are part of sum-constructions and that are not hidden, input-processes. And processes which contain atoms that are not hidden, output-processes. We want to mark them as such, so that we can try to put the input-processes at the top and the output-processes at the bottom in our animation.
From the list of atoms we can decide which are the input-processes and output-processes.

Print Graph
We start with creating a node called 'Input' to which we can connect the input-processes. Then we traverse our graph and create a node for every process that has got an ID and that is used. When we encounter a node that represents an encapsulation, we start a subgraph. If the node has a list of communications, we create edges between the processes that take part in a communication in this list. These edges are directed according to the communication. If a side of a communication is a sum-construction, it gets an arrow. Care is taken to not create multiple edges between two processes that have the same direction.
We create a node called 'Output' to which we can connect the output-processes, and we create the edges between the input and output nodes.

```graphviz
digraph ABP {
    node [color=lightblue]
    node [style=filled]
    subgraph clusterinput { I [label="Input", color=green]; }
    subgraph cluster {
        subgraph cluster1 {
            [rank=min; n4 [label="Sender"]; ]
            [rank=max; n5 [label="Receiver"]; ]
            n6 [label="K"];
            n6 -> n5 [dir=forward];
            n5 -> n6 [dir=none];
            n4 -> n6 [dir=forward];
            n7 [label="L"];
            n5 -> n7 [dir=forward];
            n4 -> n7 [dir=none];
        }
    }
    subgraph clusteroutput { O [label="Output", color=green]; }
    I -> n4 [dir=forward, label=""];
    n5 -> O [dir=forward, label=""]; 
}
```

Printing Communication List
Fupor each communication in our list, we print 'skip' if it is marked as hidden, the communication itself followed by the ID’s of the processes which cause this communication with a direction (either '-', '->', '<-', or '<->') in between.

```text
skip frame-or-error(frame(!b!, !d!)) 6 -> 5
skip frame-or-error(frame-error) 5 - 6
skip frame-comm(frame(!b!, !d!)) 4 -> 6
```
skip ack-comm(ack(!b!)) 5 -> 7
skip ack-comm(ack(!b!)) 5 -> 7
skip ack-or-error(ack(!b!)) 4 - 7
skip ack-or-error(ack-error) 4 - 7

Note that we put variable names inside ‘!’, so that we can recognize them as variables later on.

**Printing Atom List**

For each atom in our list, we print ‘skip’ if it is marked as hidden followed by the atom itself, and if it is not marked as hidden, then we print the atom followed by ‘I ->’ and the ID of the process it belongs to, if it is an input-process and the ID of the process and ‘-> O’, if it is an output-process.

```plaintext
input(!d!) I -> 4
output(!d!) 5 -> O
skip<0> 6
skip<1> 6
skip<2> 7
skip<3> 7
```

### 2.1.2 Generating the Picture

If we apply the program `dot` on the generated graph that is shown above, we get the following output (line-numbers are not part of the output).

```plaintext
digraph ABP {
node [ label = "\N",
color = lightblue,
style = filled ];
graph [lp= "81,0"];
graph [bb= "0,0,162,342"];
subgraph clusterinput {
graph [lp= ""];
graph [bb= "45,288,117,342"];
I [label=Input, color=green, pos="81,315", width="0.75", height="0.50"];}
subgraph cluster {
graph [lp= ""];
graph [bb= "0,63,162,279"];
subgraph cluster1 {
graph [bb= "9,72,153,270"];
graph [rank= min];
n4 [label=Sender, pos="81,243", width="0.81", height="0.50"];}
graph [rank= max];
n5 [label=Receiver, pos="53,154", width="0.97", height="0.50"];}
}
subgraph clusteroutput {
graph [lp= ""];
graph [bb= "14,0,92,54"]; O [label=Output, color=green, pos="53,27", width="0.83", height="0.50"];}
I -> n4 [dir=forward, pos="e,81,261 81,297 81,289 81,280 81,271"]; n4 -> n6 [dir=forward, pos="e,54,188 72,226 68,217 63,207 58,197"]; n5 -> n7 [dir=forward, pos="s,103,155 99,150 89,139 77,125 68,115"]; n4 -> n7 [dir=none, pos="90,226 95,214 103,200 108,188"];}
```

The positions are in default units, 1/72 of an inch, and widths and heights are in inches. These have to be converted to pixels, which usually are 75 per inch.

From the bounding-box in line 6, we derive the size we have to use for the window that will contain the picture. That gives us the following line.
The last part gives us a text-window of width 60 and height 10 for printing the actions that are performed in the animation.

Fupor the bounding-box in line 16 we draw a box in our picture.

We do this only for bounding-boxes belonging to a subgraph with the name cluster followed by a number. These represent the encapsulations in the specification.

We create the nodes and define for each node a text-position at which the atoms belonging to this node will be printed.

We also create the edges and define text-positions for them, at which the communications will be printed.

This results in the picture given in Figure 1.

Figure 1. Alternating Bit Protocol
Note the two lines between node $K$ and node $Receiver$. We could not determine the direction of the communication of one of them. Also, the arrow between the nodes should represent two different communications, but we found only one. Let’s take a look at the process-definitions for $Receiver$.

$Receiver = \text{Receive-Frame}(0)$

$\text{Receive-Frame}(b) = \{$

$\sum(d \in DATA, \text{receive-frame-or-error}(\text{frame}(\text{flip}(b),d)))$

$+ \text{receive-frame-or-error}(\text{frame-error})$

$\text{Send-Ack}(\text{flip}(b))$

$\sum(d \in DATA, \text{receive-frame-or-error}(\text{frame}(b,d)) \cdot$

$\text{Send-Message}(b,d)$

$\text{Send-Ack}(b) = \text{send-ack}(\text{ack}(b)) \cdot \text{Receive-Frame}(\text{flip}(b))$

$\text{Send-Message}(b,d) = \text{output}(d) \cdot \text{Send-Ack}(b)$

The three candidates to communicate here are:

receive-frame-or-error(\text{frame}(\text{flip}(b),d))

receive-frame-or-error(\text{frame-error})

receive-frame-or-error(\text{frame}(b,d))

For the first one, we cannot find a communication because of the use of the function $\text{flip}$. This function is normally rewritten, but we cannot do this statically. To inform the user, we give a warning whenever an atom is encapsulated for which we could not find a possible communication.

To solve this, we give another definition for the process $\text{Receive-Frame}$.

$\text{Receive-Frame}(b) =$

$\sum(f \in FRAME,$

receive-frame-or-error(\text{f}) . ( \$

[\text{flip(frame-bit}(f)) = b] \rightarrow \text{Send-Ack}(\text{flip}(b))$

$+ [f = \text{frame-error}] \rightarrow \text{Send-Ack}(\text{flip}(b))$

$+ [\text{frame-bit}(f) = b] \rightarrow \text{Send-Message}(b, \text{frame-data}(f))$

$\})$

We introduced here the functions $\text{frame-bit}$ and $\text{frame-data}$, which extract the concerning fields from the frame. This does not only work, it also makes the definition much clearer.

The same applies for the communications between the nodes $L$ and $Sender$, so we redefine the process $Receive-Ack$ in the same manner.

$Receive-Ack(b,d) =$

$\sum(a \in ACK,$

receive-ack-or-error(a) . ( \$

[\text{flip(ack-bit}(a)) = b] \rightarrow \text{Send-Frame}(b, d)$

$+ [a = \text{ack-error}] \rightarrow \text{Send-Frame}(b, d)$

$+ [\text{ack-bit}(a) = b] \rightarrow \text{Receive-Message}(\text{flip}(b))$

$\})$

Now we can determine the direction of the communication between $L$ and $Sender$. This results in the picture in Figure 2.

![Figure 2. Alternating Bit Protocol (adjusted)](image-url)
2.1.3 Generating the Action Function

From the list of communications and the list of atoms we derive the function which does the animation for these actions.

```tcl
proc ANIM_action {line} {
    if {{regexp {ˆskip frame-or-error\((frame\{([^,]+),([^,]+)\})\) $line}} {match}} {
        Anim::Clear n6
        Anim::CreateText txtn6ton5 "$match"
        Anim::ActivateLine linen6ton5
        Anim::AddClear n5 {line linen6ton5} {text txtn6ton5}
    } elseif {{regexp {ˆskip frame-or-error\((frame-error)\) $line}} {match}} {
        Anim::Clear n6
        Anim::Clear n5
        Anim::CreateText txtn5ton6 "$match"
        Anim::ActivateLine linen5ton6
        Anim::AddClear n5 {line linen5ton6} {text txtn5ton6}
    } elseif {{regexp {ˆskip frame-comm\((frame\{([^,]+),([^,]+)\})\) $line}} {match}} {
        Anim::Clear n4
        Anim::CreateText txtn4ton6 "$match"
        Anim::ActivateLine linen4ton6
        Anim::AddClear n6 {line linen4ton6} {text txtn4ton6}
    } elseif {{regexp {ˆskip ack-comm\((ack\{([^,]+)\})\) $line}} {match}} {
        Anim::Clear n5
        Anim::CreateText txtn5ton7 "$match"
        Anim::ActivateLine linen5ton7
        Anim::AddClear n7 {line linen5ton7} {text txtn5ton7}
    } elseif {{regexp {ˆskip ack-or-error\((ack\{([^,]+)\})\) $line}} {match}} {
        Anim::Clear n5
        Anim::CreateText txtn5ton7 "$match"
        Anim::ActivateLine linen5ton7
        Anim::AddClear n7 {line linen5ton7} {text txtn5ton7}
    } elseif {{regexp {ˆinput\((.*)\) $line}} {match}} {
        Anim::Clear I
        Anim::CreateText txtton4 "$match"
        Anim::ActivateLine lineIton4
        Anim::AddClear n4 {line lineIton4} {text txtton4}
    } elseif {{regexp {ˆoutput\((.*)\) $line}} {match}} {
        Anim::Clear n5
        Anim::CreateText txtnt0 "$match"
        Anim::ActivateLine linen5ton0
        Anim::AddClear n5 {line linen5ton0} {text txtnt0}
    } elseif {{regexp {ˆskip<0>$ line}} {match}} {
        Anim::Clear n6
        Anim::CreateText txtn6 "$match"
        Anim::AddClear n6 {text txtn6}
    } elseif {{regexp {ˆskip<1>$ line}} {match}} {
        Anim::Clear n6
        Anim::CreateText txtn6 "$match"
        Anim::AddClear n6 {text txtn6}
    } elseif {{regexp {ˆskip<2>$ line}} {match}} {
        Anim::Clear n7
        Anim::CreateText txtnt7 "$match"
        Anim::AddClear n7 {text txtnt7}
    } elseif {{regexp {ˆskip<3>$ line}} {match}} {
        Anim::Clear n7
        Anim::CreateText txtnt7 "$match"
        Anim::AddClear n7 {text txtnt7}
    }
}
```

2.1.4 Generating the Choose Function

From the list of communications and the list of atoms we also derive the function for the construction of the choose-lists for active animation. This looks the same as the action-function except for the parts inside the if-else construction.
proc ANIM_choose {line} {
    if {{[regexp {^skip frame-or-error\(frame\(.*\), \(.*\)\)\$} $line match]}} {
        Anim::AddList n6 $match
    } elseif {{[regexp {^skip frame-comm\(frame\(.*\), \(.*\)\)\$} $line match]}} {
        Anim::AddList n7 $match
    } elseif {{[regexp {^skip ack-comm\(ack\(.*\)\)\$} $line match]}} {
        Anim::AddList n4 $match
    } elseif {{[regexp {^skip ack-or-error\(ack\(.*\)\)\$} $line match]}} {
        Anim::AddList n7 $match
    } elseif {{[regexp {^input\(.*\)\$} $line match]}} {
        Anim::AddList I $match
    } elseif {{[regexp {^output\(.*\)\$} $line match]}} {
        Anim::AddList n5 $match
    } elseif {{[regexp {^skip<0>$} $line match]}} {
        Anim::AddList n6 $match
    } elseif {{[regexp {^skip<1>$} $line match]}} {
        Anim::AddList n5 $match
    } elseif {{[regexp {^skip<2>$} $line match]}} {
        Anim::AddList n7 $match
    } elseif {{[regexp {^skip<3>$} $line match]}} {
        Anim::AddList n7 $match
    }
}

2.2 More on Process Graphs

2.2.1 Merge

In order to show how we deal with the generalized merge, we consider a specification of a small factory consisting of six stations connected by conveyor belts, with an input and an output. It produces two products which take slightly different routes through the factory. The complete specification can be found in A.2. Here we show the process definitions for the stations.

```tcl
Stations = merge(s in STATION-set, Station(s))
Station(s) =
    [eq-stat(s, 1) = true] ->
        sum(p in PRODUCT, read-input(p) . to-belt(s, next(s, p), p)) . Station(s)
    + [eq-stat(s, 6) = true] ->
        sum(p in PRODUCT, from-belt(s, p) . send-output(p)) . Station(s)
    + [and(not(eq-stat(s, 1)), not(eq-stat(s, 6))) = true] ->
        sum(p in PRODUCT, from-belt(s, p) . to-belt(s, next(s, p), p)) . Station(s)
```

If we simply expand the merge as many times as there are elements in the set STATION-set, we end up with six stations that can all communicate with each other. But we want only the communications that really represent a conveyor belt. We could do a better job if the conditional expressions do not contain a variable, so we can evaluate them and disregard the following process expression on a negative result.

So, we have to expand the merge for each element of the set with this element filled in for the variable of the sum operator, and replace every occurrence of a variable with its value, whether it is a variable of a sum operator, or a variable we obtained a value for from matching a process with the left hand side of a process definition.

We give here the equations for the function next which decides what the next station is.

[3] next(1, p) = 2
[4] next(2, p) = 3
We see that a rewriting of the function next with only a value given for the station, gives us the new station, except for station 3 since it depends on the product. We can alter the last part of the process definition like this.

```plaintext
+ [and(not(eq-stat(s, 1)), not(eq-stat(s, 6))) = true] -> (sum(p in PRODUCT, from-belt(s, p) . (p = A) -> to-belt(s, next(s, A), p) + p = B) -> to-belt(s, next(s, B), p) ) . Station(s)
)
```

This gives us the picture in Figure 3.

2.2.2 Combination of Processes

Consider now a generalized form of the factory in which all stations are connected with each other by conveyor belts. We use a scheduler to control this factory in such a way that it acts the same as the factory in the previous factory. The specification can be found in A.3.

Let's take a look at the specification of the scheduler.

```plaintext
Scheduler = sum(p in PRODUCT, rec-start(p) . (SubScheduler(1, p) || Scheduler))
SubScheduler(s, p) = [not(eq-stat(s, 6)) = true] -> (rec-request(s, p) . Next(s, p, next(s, p)) ) + [s = 6] -> rec-end
Next(s, p, n) = send-next(s, n) . SubScheduler(n, p)
```

We see here that for each product a subscheduler is created. If we generate an animation for this specification it gives us the picture in Figure 4.

The processes Scheduler and SubScheduler in this picture do not reflect the specification. We should create
and destroy \textit{SubScheduler} processes dynamically, but that is not possible (at the moment). But since there is no communication possible between the \textit{Scheduler} and \textit{SubScheduler}, or between two \textit{SubSchedulers}, we can consider them as one process. This results in the picture shown in Figure 5.

![Diagram of a scheduled factory with combined processes]

\textbf{Figure 5.} scheduled factory with combined processes

Whether this behavior is always wanted, we do not know, so we made this combination of processes optional.

\subsection*{2.3 Remarks}

Although it seems that we can generate animations for all specifications with only a few adjustments, we should keep in mind that expanding processes is done through open term matching with the left hand side of process definition. This can result in a mismatch since the process to expand may have an argument that should be rewritten in order to match but contains a variable which prevents a rewrite.

Also, deciding if an atom is an element of a hide or an encapsulation set is open term matching and thus can result in a mismatch for the same reason.

So we must try to circumvent these situations. We can use conditional expressions for this, but they make the specifications larger.

The direction of a communication is now based on the presence of a sum-construction at the sides of the communication. In some cases, we could try to do a better job by examining the context of both sides of the communication.

We should also note that a sum-construction is not always meant to be a port. It could for instance also be used to connect to a random process.

Despite the above, generating an animation is very useful in testing and understanding specifications. One of its main advantages is that a generated animation reflects the specification, in contrast with other techniques such as visualization through transition systems, so that events can easily be traced back to their origin in the specification.

\section*{Acknowledgements}

Thanks to Alban Ponse for his proofreading and remarks.

\section*{3. References}


A. PSF Specifications

A.1 Alternating Bit Protocol

```
data module Bits
begin
  exports
  begin
    sorts
      BIT
    functions
      0 :-> BIT
      1 :-> BIT
      flip : BIT -> BIT
  end
  equations
    [B1] flip(0) = 1
    [B2] flip(1) = 0
end Bits
```

```
data module Data
begin
  exports
  begin
    sorts
      DATA
    functions
      'a :-> DATA
      'b :-> DATA
      'c :-> DATA
      'd :-> DATA
      'e :-> DATA
  end
end Data
```

```
data module Frames
begin
  exports
  begin
    sorts
      FRAME
    functions
      frame : BIT # DATA -> FRAME
      frame-error :-> FRAME
  end
  imports
    Data, Bits
end Frames
```

```
data module Acknowledgements
begin
  exports
  begin
    sorts
      ACK
    functions
      ack : BIT -> ACK
      ack-error :-> ACK
  end
  imports
    Bits
end Acknowledgements
```

```
process module ABP
begin
  imports
    Bits, Data, Frames, Acknowledgements
  atoms
    input : DATA
    send-frame : FRAME
    receive-ack-or-error : ACK
    receive-frame : FRAME
    send-frame-or-error : FRAME
    receive-frame-or-error : FRAME
    output : DATA
    send-ack : ACK
    receive-ack : ACK
    send-ack-or-error : ACK
    frame-comm : FRAME
```
frame-or-error : FRAME
ack-comm : ACK
ack-or-error : ACK

processes
Sender
Receive-Message : BIT
Send-Frame : BIT # DATA
Receive-Ack : BIT # DATA
K
K : BIT # DATA
Receiver
Receive-Frame : BIT
Send-Ack : BIT
Send-Message : BIT # DATA
L
L : BIT

ABP

sets
of atoms
H = { send-frame(f), receive-frame(f) | f in FRAME }
+ { send-frame-or-error(f), receive-frame-or-error(f) | f in FRAME }
+ { send-ack(a), receive-ack(a) | a in ACK }
+ { send-ack-or-error(a), receive-ack-or-error(a) | a in ACK }
I = { frame-comm(f), frame-or-error(f) | f in FRAME }
+ { ack-comm(a), ack-or-error(a) | a in ACK }

of BIT
Bit-Set = { 0, 1 }

communications
send-frame(f) | receive-frame(f) = frame-comm(f) for f in FRAME
send-frame-or-error(f) | receive-frame-or-error(f) = frame-or-error(f) for f in FRAME
send-ack(a) | receive-ack(a) = ack-comm(a) for a in ACK
send-ack-or-error(a) | receive-ack-or-error(a) = ack-or-error(a) for a in ACK

variables
f ::= FRAME
b ::= BIT
d ::= DATA
a ::= ACK

definitions
Sender = Receive-Message(0)
Receive-Message(b) = sum(d in DATA, input(d) . Send-Frame(b,d))
Send-Frame(b,d) = send-frame(frame(b,d)) . Receive-Ack(b,d)
Receive-Ack(b,d) = {
  receive-ack-or-error(ack(flip(b)))
  + receive-ack-or-error(ack-error)
} . Send-Frame(b,d)
  + receive-ack-or-error(ack(b)) . Receive-Message(flip(b))
K = sum(d in DATA, sum(b in Bit-Set, receive-frame(frame(b,d)) . K(b,d)))
K(b,d) = {
  skip . send-frame-or-error(frame(b,d))
  + skip . send-frame-or-error(frame-error)
} . K

Receiver = Receive-Frame(0)
Receive-Frame(b) = {
  sum(d in DATA, receive-frame-or-error(frame(flip(b),d)))
  + receive-frame-or-error(frame-error)
} . Send-Ack(flip(b))
Send-Ack(b) = send-ack(ack(b)) . Receive-Frame(flip(b))
Send-Message(b,d) = output(d) . Send-Ack(b)
L = sum(b in Bit-Set, receive-ack(ack(b)) . L(b))
L(b) = {
  skip . send-ack-or-error(ack(b))
  + skip . send-ack-or-error(ack-error)
} . L

ABP = hide(I, encaps(H, Sender || Receiver || K || L))

end ABP

A.2 Factory
data module Products begin
exports begin
  sorts
    PRODUCT
  functions
    A : -> PRODUCT
    B : -> PRODUCT
end
end Products

data module Stations
begin
exports begin
  sorts
    STATION
  functions
    1 : -> STATION
    2 : -> STATION
    3 : -> STATION
    4 : -> STATION
    5 : -> STATION
    6 : -> STATION
    eq-stat : STATION # STATION -> BOOLEAN
    next : STATION # PRODUCT -> STATION
end imports
  Booleans, Products
variables
  x : -> STATION
  y : -> STATION
  p : -> PRODUCT
equations
  [1] eq-stat(x, x) = true
  [2] not(eq-stat(x, y)) = true
  [3] next(1, p) = 2
  [4] next(2, p) = 3
  [5] next(3, A) = 4
  [6] next(3, B) = 5
  [7] next(4, p) = 5
  [8] next(5, p) = 6
end Stations

process module Factory
begin
imports
  Stations
atoms
  input : PRODUCT
  output : PRODUCT
  read-input : PRODUCT
  send-input : PRODUCT
  comm-input : PRODUCT
  read-output : PRODUCT
  send-output : PRODUCT
  comm-output : PRODUCT
  to-belt : STATION # STATION # PRODUCT
  from-belt : STATION # PRODUCT
  comm-belt : STATION # STATION # PRODUCT
processes
  Start
  Input
  Stations
  Station : STATION
  Output
sets
  of PRODUCT
    PRODUCT-set = { A, B }
  of STATION
    STATION-set = { 1, 2, 3, 4, 5, 6 }
  of atoms
    H = { send-input(p), read-input(p), send-output(p), read-output(p),
         to-belt(x, y, p), from-belt(y, p) | p in PRODUCT,
         x in STATION, y in STATION }
communications
  send-input(p) | read-input(p) = comm-input(p)
  for p in PRODUCT
  send-output(p) | read-output(p) = comm-output(p)
  for p in PRODUCT
  to-belt(s1, s2, p) | from-belt(s2, p) = comm-belt(s1, s2, p)
  for s1 in STATION, s2 in STATION, p in PRODUCT
variables
  s : -> STATION
definitions
Start = $\text{encaps}(H, \text{Input} \ || \ \text{Stations} \ || \ \text{Output})$
Input = $\sum (p \in \text{PRODUCT}-set, \text{input}(p) \cdot \text{send-input}(p)) \cdot \text{Input}$
Stations = $\text{merge}(s \in \text{STATION}-set, \text{Station}(s))$
Station(s) =
$\begin{cases}
\text{[eq-stat}(s, 1) = \text{true}] & \rightarrow \{\\
\sum (p \in \text{PRODUCT}, \text{read-input}(p) \cdot \text{to-belt}(s, \text{next}(s, p), p) \\
\} \cdot \text{Station}(s)\\
\end{cases}$

$\begin{cases}
\text{[eq-stat}(s, 6) = \text{true}] & \rightarrow \{\\
\sum (p \in \text{PRODUCT}, \text{from-belt}(s, p) \cdot \text{send-output}(p) \\
\} \cdot \text{Station}(s)\\
\end{cases}$

$\begin{cases}
\text{[and(not(eq-stat}(s, 1)), not(eq-stat}(s, 6)) = \text{true}] & \rightarrow \{\\
\sum (p \in \text{PRODUCT}, \text{from-belt}(s, p) \cdot \text{to-belt}(s, \text{next}(s, p), p) \\
\} \cdot \text{Station}(s)\\
\end{cases}$
Output = $\sum (p \in \text{PRODUCT}, \text{read-output}(p) \cdot \text{output}(p)) \cdot \text{Output}$
end Factory

A.3 Scheduled Factory

Imported modules not given here, are the same as from the factory without scheduler.

process module Scheduler
begin
exports
begin
atoms
send-request : \text{STATION} \ # \ \text{PRODUCT}
rec-request : \text{STATION} \ # \ \text{PRODUCT}
comm-request : \text{STATION} \ # \ \text{PRODUCT}
send-next : \text{STATION} \ # \ \text{STATION}
rec-next : \text{STATION} \ # \ \text{STATION}
comm-next : \text{STATION} \ # \ \text{STATION}
send-start : \text{PRODUCT}
rec-start : \text{PRODUCT}
comm-start : \text{PRODUCT}
send-end
rec-end
comm-end
processes
Scheduler
sets
of atoms
HS = \{ send-request(s1, p), rec-request(s1, p), 
send-next(s1, s2), rec-next(s1, s2), 
send-start(p), rec-start(p), send-end, rec-end 
| s1 \in \text{STATION}, s2 \in \text{STATION}, p \in \text{PRODUCT} \}
end imports
Stations
processes
Next : \text{STATION} \ # \ \text{PRODUCT} \ # \ \text{STATION}
SubScheduler : \text{STATION} \ # \ \text{PRODUCT}
communications
send-request(s, p) \ | \ rec-request(s, p) = \text{comm-request}(s, p)
for s \in \text{STATION}, p \in \text{PRODUCT}
send-next(s1, s2) \ | \ rec-next(s1, s2) = \text{comm-next}(s1, s2)
for s1 \in \text{STATION}, s2 \in \text{STATION}
send-start(p) \ | \ rec-start(p) = \text{comm-start}(p)
for p \in \text{PRODUCT}
send-end \ | \ rec-end = \text{comm-end}
variables
s : \rightarrow \text{STATION}
n : \rightarrow \text{STATION}
p : \rightarrow \text{PRODUCT}
definitions
Scheduler =
$\begin{cases}
\sum (p \in \text{PRODUCT}, 
\text{rec-start}(p) \ . \\
( \text{SubScheduler}(1, p) \\
\{\}} \\
\} \\
\) \\
\) \\
SubScheduler(s, p) = \$
\[ \text{not}(eq-stat(s, 6)) = true \] -> ( 
  rec-request(s, p) .
  Next(s, p, next(s, p))
) + \[ s = 6 \] ->
  rec-end

Next(s, p, n) = send-next(s, n) .
SubScheduler(n, p)
end

Scheduler

process module Factory
begin
inputs
  Stations,
  Scheduler
atoms
  input : PRODUCT
  output : PRODUCT
  read-input : PRODUCT
  send-input : PRODUCT
  comm-input : PRODUCT
  read-output : PRODUCT
  send-output : PRODUCT
  comm-output : PRODUCT
  to-belt : STATION # STATION # PRODUCT
  from-belt : STATION # PRODUCT
  comm-belt : STATION # STATION # PRODUCT

processes
  Start
  Input
  Stations
  Station : STATION
  Output
sets
  of PRODUCT
    PRODUCT-set = { A, B }
  of STATION
    STATION-set = { 1, 2, 3, 4, 5, 6 }
  of atoms
    \( H = \{ \text{send-input}(p), \text{read-input}(p), \text{send-output}(p), \text{read-output}(p), \text{to-belt}(x, y, p), \text{from-belt}(y, p) | p \in \text{PRODUCT}, x \in \text{STATION}, y \in \text{STATION} \} \)
communications
  send-input(p) | read-input(p) = comm-input(p)
  for p in PRODUCT
  send-output(p) | read-output(p) = comm-output(p)
  for p in PRODUCT
  to-belt(s1, s2, p) | from-belt(s2, p) = comm-belt(s1, s2, p)
  for s1 in STATION, s2 in STATION, p in PRODUCT
variables
  s : \rightarrow \text{STATION}
definitions
  Start = \text{encaps}(H, \text{Scheduler} \mid \mid \text{encaps}(H, \text{Input} \mid \mid \text{Stations} \mid \mid \text{Output}))
  Input = \text{sum}(p \in \text{PRODUCT-set},
    \text{input}(p) .
    \text{send-start}(p) .
    \text{send-input}(p))
  . \text{Input}
  Stations = \text{merge}(s \in \text{STATION-set}, \text{Station}(s))
Stations(s) =
  \[ \text{eq-stat}(s, 1) = true \] -> ( 
    \text{sum}(p \in \text{PRODUCT},
      \text{read-input}(p) .
      \text{send-request}(s, p) .
      \text{sum}(n \in \text{STATION},
        \text{rec-next}(s, n) .
        \text{to-belt}(s, n, p))
    ) . \text{Station}(s)
  ) + \[ \text{eq-stat}(s, 6) = true \] -> ( 
    \text{sum}(p \in \text{PRODUCT},
      \text{from-belt}(s, p) .
      \text{send-output}(p))
  ) . \text{Station}(s)
  + \[ \text{and}(\text{not}(\text{eq-stat}(s, 1)), \text{not}(\text{eq-stat}(s, 6))) = true \] -> ( 
    \text{sum}(p \in \text{PRODUCT},
      \text{from-belt}(s, p) .
      \text{send-request}(s, p) .
      \text{sum}(n \in \text{STATION},
rec-next(s, n) .
to-belt(s, n, p)
)
)
). Station(s)
)
Output =
\sum_{p \in \text{PRODUCT}}
read-output(p) .
send-end .
output(p)
)
). Output
end Factory
Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (http://www.science.uva.nl/research/prog/reports/) or by anonymous ftp to ftp.science.uva.nl directory pub/programming-research/reports/.


[P9720] M. van der Graaf. A Specification of Box to HTML in ASF+SDF.


[P9610] T.B. Dinesh and S.M. Üsküdarlı. Specifying input and output of visual languages

[P9609] T.B. Dinesh and S.M. Üsküdarlı. The VAS formalism in VASE.


[P9601] P.A. Olivier. Embedded system simulation: testdriving the ToolBus

[P9512] J.J. Brunekeef. TransLog, an interactive tool for transformation of logic programs


[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*


[P9420] M.G.J. van den Brand and E. Visser. *From Box to TeX: An algebraic approach to the construction of documentation tools.*


[P9328] B. Dietens. *A simulator for PSF in PSF.*


[P9301] J.J. van Wam. *A library for PSF.*

[P9208b] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208).*


[P9005b] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra (revised version).*


[P9212] J.J. van Wam. *A study of a one bit sliding window protocol in ACP.*


[P9205] J.J. van Wamel. *An algebraic verification of the concurrent alternating bit protocol*


[P9203] J.A. Verschuren. *A simulator for mCRL in ASF+SDF.*


[P9105] C. Verhoef. *An operator definition principle (for process algebras).*


[P9009] G.J. Veltink. *From PSF to TIL.*


[P9002b] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero a object (revised version of P9002).*


formalism based on static COLD (revised version of P8906).


[P8916b] J.C.M. Baeten and J.A. Bergstra. Real time process algebra (revised version of
P8916).

[P9002] J.C.M. Baeten and J.A. Bergstra. Process algebra with zero object and non-
determinacy.


syntax engineering (preliminary version).


[P8808b] J.A. Bergstra. A mode transfer operator in process algebra (revised version of
P8808).


formalism based on static COLD.

(revised version of P8815).

[P8905] J.A. Bergstra and J.W. Klop. BMACP.

sorts with equality.


[P8819] H.K. Faber. Strategische uitleg in een medisch expertsysteem.
[P8802] J.C.M. Baeten and J.A. Bergstra. Recursive process definitions with the state operator.


[P8703] A.V. Hurkmans. *Een declaratieve en procedurele kennisrepresentatievorm voor kennisystemen, toegepast op NEXT.*
