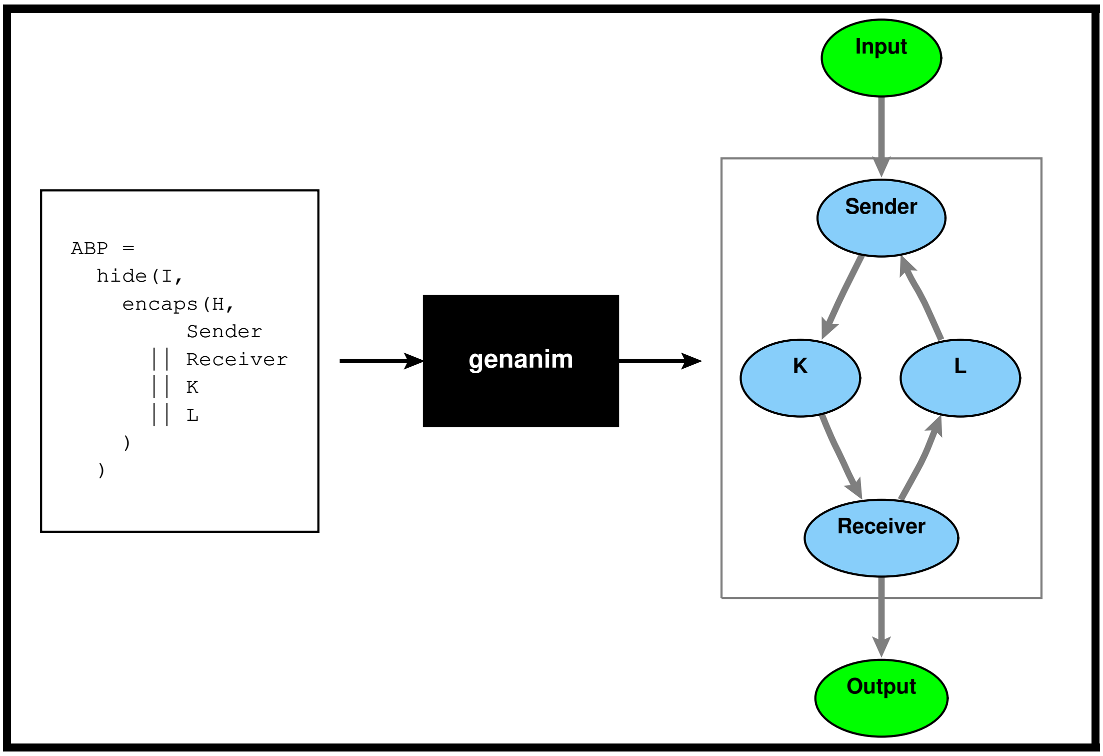
University of Amsterdam

Programming Research Group



Generation of Animations for Simulation of Process Algebra Specifications

Bob Diertens

Report P0003 october 2000

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

B. Diertens

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7593
e-mail: bobd@science.uva.nl

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 extend 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.

```
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.

```
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

Fupor each atom in our list, we print 'skip' if it is marked as hidden followed by the atom itself, and if it 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.

```
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).

```
 1 digraph ABP {
 2    node [   label = "\N",
 3       color = lightblue,
 4       style = filled ];
 5    graph [lp= "81,0"];
 6    graph [bb= "0,0,162,342"];
 7    subgraph clusterinput {
 8       graph [lp= ""];
 9       graph [bb= "45,288,117,342"];
10       I [label=Input, color=green, pos="81,315", width="0.75", height="0.50"];
11    }
12    subgraph cluster {
13       graph [lp= ""];
14       graph [bb= "0,63,162,279"];
15       subgraph cluster1 {
16          graph [bb= "9,72,153,270"];
17          {
18             graph [rank= min];
19             graph [bb= ""];
20             n4 [label=Sender, pos="81,243", width="0.81", height="0.50"];
21          }
22          {
23             graph [rank= max];
24             graph [bb= ""];
25             n5 [label=Receiver, pos="53,99", width="0.97", height="0.50"];
26          }
27          n6 [label=K, pos="45,171", width="0.75", height="0.50"];
28          n7 [label=L, pos="117,171", width="0.75", height="0.50"];
29          n6 -> n5 [dir=forward, pos="e,45,117 41,153 41,145 42,135 43,127"];
30          n5 -> n6 [dir=none, pos="53,154 55,143 57,128 57,117"];
31          n4 -> n6 [dir=forward, pos="e,54,188 72,226 68,217 63,207 58,197"];
32          n5 -> n7 [dir=forward, pos="s,103,155 99,150 89,139 77,125 68,115"];
33          n4 -> n7 [dir=none, pos="90,226 95,214 103,200 108,188"];
34       }
35    }
36    subgraph clusteroutput {
37       graph [lp= ""];
38       graph [bb= "14,0,92,54"];
39       O [label=Output, color=green, pos="53,27", width="0.83", height="0.50"];
40    }
41    I -> n4 [dir=forward, pos="e,81,261 81,297 81,289 81,280 81,271"];
42    n5 -> O [dir=forward, pos="e,53,45 53,81 53,73 53,64 53,55"];
43 }
```

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.

```
Anim::Windows 188 376 -text 60 10
```

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.

```
Anim::CreateLine box1 pos 19 291 pos 169 291 pos 169 85 pos 19 85 pos 19 291 -width 1
```

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.

```
Anim::CreateItem I oval 94 38 28 18 "Input" -anchor s -color 1
Anim::TextposItem textI I ce n
Anim::CreateItem n4 oval 94 113 30 18 "Sender" -anchor s -color 0
Anim::TextposItem textn4 n4 ce n
Anim::CreateItem n5 oval 65 263 36 18 "Receiver" -anchor s -color 0
Anim::TextposItem textn5 n5 ce n
Anim::CreateItem n6 oval 56 188 28 18 "K" -anchor s -color 0
Anim::TextposItem textn6 n6 ce n
Anim::CreateItem n7 oval 131 188 28 18 "L" -anchor s -color 0
Anim::TextposItem textn7 n7 ce n
Anim::CreateItem O oval 65 338 31 18 "Output" -anchor s -color 1
Anim::TextposItem textO O ce n
```

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

```
Anim::CreateLine linen6ton5 pos 52 206 pos 52 215 pos 53 225 pos 54 233 pos 56 244 \
   -arrow last -smooth
Anim::Textpos textn6ton5 53 225 ce
Anim::CreateLine linen5ton6 pos 65 205 pos 67 217 pos 69 232 pos 69 244 -arrow none \
   -smooth
Anim::Textpos textn5ton6 68 224 ce
Anim::CreateLine linen4ton6 pos 85 130 pos 80 140 pos 75 150 pos 70 161 pos 66 170 \
   -arrow last -smooth
Anim::Textpos textn4ton6 75 150 ce
Anim::CreateLine linen5ton7 pos 117 204 pos 113 210 pos 102 221 pos 90 236 pos 80 246 \
   -arrow first -smooth
Anim::Textpos textn5ton7 102 221 ce
Anim::CreateLine linen4ton7 pos 103 130 pos 108 143 pos 117 157 pos 122 170 -arrow none \
   -smooth
Anim::Textpos textn4ton7 112 150 ce
Anim::CreateLine lineIton4 pos 94 56 pos 94 65 pos 94 74 pos 94 83 pos 94 94 -arrow last \
   -smooth
Anim::Textpos textIton4 94 74 ce
Anim::CreateLine linen5toO pos 65 281 pos 65 290 pos 65 299 pos 65 308 pos 65 319 \
   -arrow last -smooth
Anim::Textpos textn5toO 65 299 ce
```

This results in the picture given in Figure 1.

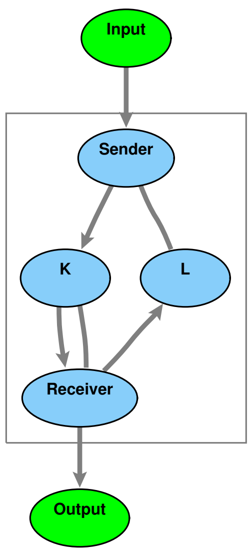


Figure 1. Alternating Bit Protocol

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. Lets take a look at the process-definitions for *Receiver*.

```
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))
        + sum(d in DATA, receive-frame-or-error(frame(b,d)) .
            Send-Message(b,d)
          )
Send-Ack(b) = send-ack(ack(b)) . Receive-Frame(flip(b))
Send-Message(b,d) = output(d) . Send-Ack(b)
```

The three candidates to communicate here are:

```
receive-frame-or-error(frame(flip(b),d))
receive-frame-or-error(frame-error)
receive-frame-or-error(frame(b,d))
```

Fupor the first one, we can not find a communication because of the use of the function *flip*. This function is normally rewritten, but we can not 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 *Receive-Frame*.

```
Receive-Frame(b) =
      sum(f in FRAME,
          receive-frame-or-error(f) . (
             [flip(frame-bit(f)) = b]-> Send-Ack(flip(b))
          +  [f = frame-error]-> Send-Ack(flip(b))
          +  [frame-bit(f) = b]-> Send-Message(b, frame-data(f))
          )
      )
```

We introduced here the functions *frame-bit* and *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) . (
             [flip(ack-bit(a)) = b]-> Send-Frame(b, d)
          +  [a = ack-error]-> Send-Frame(b, d)
          +  [ack-bit(a) = b]-> Receive-Message(flip(b))
          )
      )
```

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

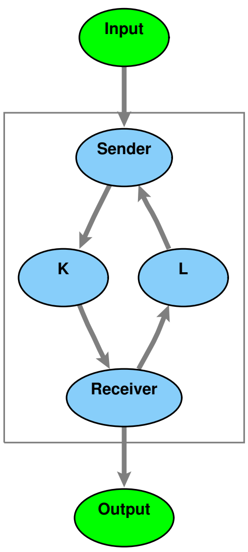


Figure 2. Alternating Bit Protocol (adjusted)

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.

```
proc ANIM_action {line} {
   if {[regexp {^skip frame-or-error\(frame\((.*), (.*)\)\)$} $line match]} {
      Anim::Clear n6
      Anim::CreateText textn6ton5 "$match"
      Anim::ActivateLine linen6ton5
      Anim::AddClear n5 {line linen6ton5} {text textn6ton5}
   } elseif {[regexp {^skip frame-or-error\(frame-error\)$} $line match]} {
      Anim::Clear n6
      Anim::Clear n5
      Anim::CreateText textn5ton6 "$match"
      Anim::ActivateLine linen5ton6
      Anim::AddClear n5 {line linen5ton6} {text textn5ton6}
      Anim::AddClear n6 {line linen5ton6} {text textn5ton6}
   } elseif {[regexp {^skip frame-comm\(frame\((.*), (.*)\)\)$} $line match]} {
      Anim::Clear n4
      Anim::CreateText textn4ton6 "$match"
      Anim::ActivateLine linen4ton6
      Anim::AddClear n6 {line linen4ton6} {text textn4ton6}
   } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
      Anim::Clear n5
      Anim::CreateText textn5ton7 "$match"
      Anim::ActivateLine linen5ton7
      Anim::AddClear n7 {line linen5ton7} {text textn5ton7}
   } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
      Anim::Clear n5
      Anim::CreateText textn5ton7 "$match"
      Anim::ActivateLine linen5ton7
      Anim::AddClear n7 {line linen5ton7} {text textn5ton7}
   } elseif {[regexp {^skip ack-or-error\(ack\((.*)\)\)$} $line match]} {
      Anim::Clear n7
      Anim::Clear n4
      Anim::CreateText textn4ton7 "$match"
      Anim::ActivateLine linen4ton7
      Anim::AddClear n4 {line linen4ton7} {text textn4ton7}
      Anim::AddClear n7 {line linen4ton7} {text textn4ton7}
   } elseif {[regexp {^skip ack-or-error\(ack-error\)$} $line match]} {
      Anim::Clear n7
      Anim::Clear n4
      Anim::CreateText textn4ton7 "$match"
      Anim::ActivateLine linen4ton7
      Anim::AddClear n4 {line linen4ton7} {text textn4ton7}
      Anim::AddClear n7 {line linen4ton7} {text textn4ton7}
   } elseif {[regexp {^input\((.*)\)$} $line match]} {
      Anim::Clear I
      Anim::CreateText textIton4 "$match"
      Anim::ActivateLine lineIton4
      Anim::AddClear n4 {line lineIton4} {text textIton4}
   } elseif {[regexp {^output\((.*)\)$} $line match]} {
      Anim::Clear n5
      Anim::CreateText textn5toO "$match"
      Anim::ActivateLine linen5toO
      Anim::AddClear n5 {line linen5toO} {text textn5toO}
   } elseif {[regexp {^skip<0>$} $line match]} {
      Anim::Clear n6
      Anim::CreateText textn6 "$match"
      Anim::AddClear n6 {text textn6}
   } elseif {[regexp {^skip<1>$} $line match]} {
      Anim::Clear n6
      Anim::CreateText textn6 "$match"
      Anim::AddClear n6 {text textn6}
   } elseif {[regexp {^skip<2>$} $line match]} {
      Anim::Clear n7
      Anim::CreateText textn7 "$match"
      Anim::AddClear n7 {text textn7}
   } elseif {[regexp {^skip<3>$} $line match]} {
      Anim::Clear n7
      Anim::CreateText textn7 "$match"
      Anim::AddClear n7 {text textn7}
   }
}
```

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-or-error\(frame-error\)$} $line match]} {
      Anim::AddList n6 $match
      Anim::AddList n5 $match
   } elseif {[regexp {^skip frame-comm\(frame\((.*), (.*)\)\)$} $line match]} {
      Anim::AddList n4 $match
   } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
      Anim::AddList n5 $match
   } elseif {[regexp {^skip ack-comm\(ack\((.*)\)\)$} $line match]} {
      Anim::AddList n5 $match
   } elseif {[regexp {^skip ack-or-error\(ack\((.*)\)\)$} $line match]} {
      Anim::AddList n7 $match
      Anim::AddList n4 $match
   } elseif {[regexp {^skip ack-or-error\(ack-error\)$} $line match]} {
      Anim::AddList n7 $match
      Anim::AddList n4 $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 n6 $match
   } elseif {[regexp {^skip<2>$} $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 conveyer 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.

```
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 conveyer 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
```

```
[5] next(3, A) = 4
[6] next(3, B) = 5
[7] next(4, p) = 5
[8] next(5, p) = 6
```

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.

```
+  [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.

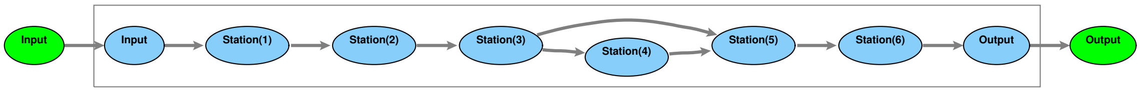


Figure 3. factory

We used here an option that gives an orientation from left to right, instead of the default top to bottom.

2.2.2 Combination of Processes

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

Lets take a look at the specification of the scheduler.

```
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.

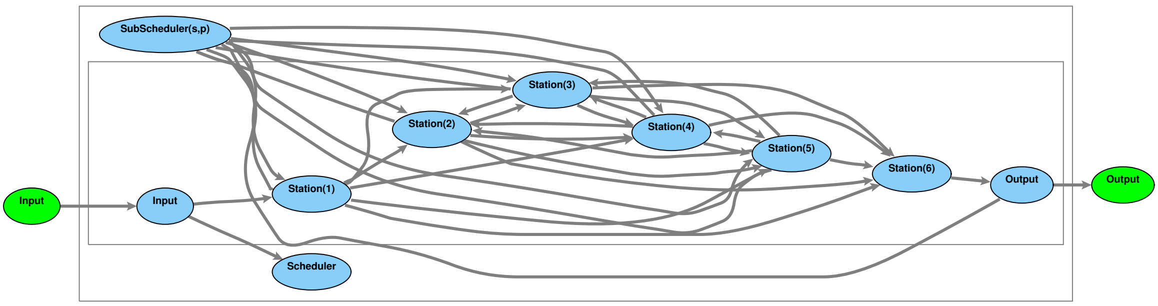


Figure 4. scheduled factory

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

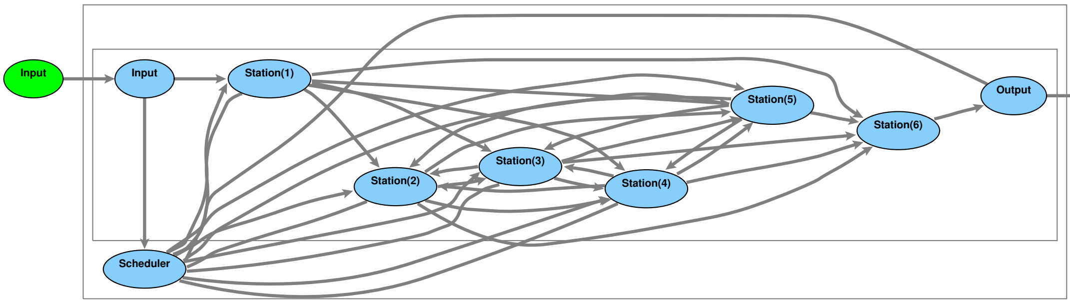


Figure 5. scheduled factory with combined processes

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

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.

Acknowledgements

Thanks to Alban Ponse for his proofreading and remarks.

3. References

[Die94] B. Diertens, "New Features in PSF I - Interrupts, Disrupts, and Priorities," report P9417, Programming Research Group - University of Amsterdam, June 1994.

[Die97] B. Diertens, "Simulation and Animation of Process Algebra Specifications," report P9713, Programming Research Group - University of Amsterdam, September 1997.

[DiePon94] B. Diertens and A. Ponse, "New Features in PSF II - Iteration and Nesting," report P9425, Programming Research Group - University of Amsterdam, October 1994.

[MauVel90] S. Mauw and G.J. Veltink, "A Process Specification Formalism," in *Fundamenta Informaticae XIII (1990)*, pp. 85-139, IOS Press, 1990.

[MauVel93] S. Mauw and G.J. Veltink (eds.), *Algebraic Specification of Communication Protocols,* Cambridge Tracts in Theoretical Computer Science 36, Cambridge University Press, 1993.

[WalChrSch96] L. Wall, T. Christiansen, and R.L. Schwartz, *Programming Perl,* O'Reilly & Associates, Inc., 1996.

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))
            +   sum(d in DATA, receive-frame-or-error(frame(b,d)) .
                    Send-Message(b,d)
                )
        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 = encaps(H, Input || Stations || Output)
        Input = sum(p in PRODUCT-set, input(p) . send-input(p)) . Input
        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)
                )
        Output = sum(p in PRODUCT, read-output(p) . output(p)) . 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 : STATION # PRODUCT
            rec-request : STATION # PRODUCT
            comm-request : STATION # PRODUCT
            send-next : STATION # STATION
            rec-next : STATION # STATION
            comm-next : STATION # STATION
            send-start : PRODUCT
            rec-start : PRODUCT
            comm-start : 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 STATION, s2 in STATION, p in PRODUCT }
    end
    imports
        Stations
    processes
        Next : STATION # PRODUCT # STATION
        SubScheduler : STATION # PRODUCT
    communications
        send-request(s, p) | rec-request(s, p) = comm-request(s, p)
            for s in STATION, p in PRODUCT
        send-next(s1, s2) | rec-next(s1, s2) = comm-next(s1, s2)
            for s1 in STATION, s2 in STATION
        send-start(p) | rec-start(p) = comm-start(p)
            for p in PRODUCT
        send-end | rec-end = comm-end
    variables
        s : -> STATION
        n : -> STATION
        p : -> PRODUCT
    definitions
        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)
end Scheduler

process module Factory
begin
   imports
      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 = { 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 = encaps(HS, Scheduler || encaps(H, Input || Stations || Output))
      Input =
         sum(p in PRODUCT-set,
            input(p) .
            send-start(p) .
            send-input(p)
         ) . Input
      Stations = merge(s in STATION-set, Station(s))
      Station(s) =
            [eq-stat(s, 1) = true]-> (
               sum(p in PRODUCT,
                  read-input(p) .
                  send-request(s, p) .
                  sum(n in STATION,
                     rec-next(s, n) .
                     to-belt(s, n, 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) .
                  send-request(s, p) .
                  sum(n in STATION,
```

```
                    rec-next(s, n) .
                    to-belt(s, n, p)
                  )
                ) . Station(s)
              )
          Output =
            sum(p in 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/`.

[P0003] B. Diertens. *Generation of Animations for Simulation of Process Algebra Specifications.*

[P0002] J.A. Bergstra, I. Bethke, and M.E. Loots. *A First Course on C Program Reading.*

[P0001] J.A. Bergstra, I. Bethke, and A. Ponse. *Basic Multi-Competence Programming.*

[P9906] J.A. Bergstra and M.E. Loots. *Software mechanics for Java multi-threading.*

[P9905] M.P.A. Sellink and C. Verhoef. *Towards Automated Modification of Legacy Assets.*

[P9904] M.P.A. Sellink and C. Verhoef. *Scaffolding for Software Renovation.*

[P9903] René Krikhaar, André Postma, M.P.A. Sellink, Marc Stroucken, and C. Verhoef. *A Two-phase Process for Software Architecture Improvement.*

[P9902] A. van Deursen, P. Klint, and C. Verhoef. *Research Issues in the Renovation of Legacy Systems.*

[P9901] M.P.A. Sellink and C. Verhoef. *Generation of Software Renovation Factories from Compilers.*

[P9811] J.A. Bergstra and M.E. Loots. *Program Algebra for Component Code.*

[P9810] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. *Restructuring of* COBOL/CICS *Legacy Systems.*

[P9809] B. Wedemeijer. *Introduction & Basic Tooling for* CASL *using* ASF+SDF.

[P9808] J.A. Bergstra and A. Ponse. *Two Recursive Generalizations of Iteration in Process Algebra.*

[P9807] M.P.A. Sellink and C. Verhoef. *An Architecture for Software Maintenance.*

[P9806] M. B. van der Zwaag. *Some Verifications in Process Algebra with Iota.*

[P9805] M.P.A. Sellink and C. Verhoef. *Development, Assessment, and Reengineering of Language Descriptions.*

[P9804] M.P.A. Sellink and C. Verhoef. *Native Patterns.*

[P9803] J.J. Brunekreef. *Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language* Alma-0.

[P9802] W. Fokkink and C. Verhoef. *Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories.*

[P9801] P. Klint and C. Verhoef. *Evolutionary Software Engineering: A Component-based Approach.*

[P9726] J.A. Bergstra and A. Ponse. *Grid Protocol Specifications.*

[P9725] J.A. Bergstra and A. Ponse. *Process Algebra Primitives for File Transfer.*

[P9724] J.A. Bergstra and A. Ponse. *Process Algebra with Four-Valued Logic.*

[P9723] J.A. Bergstra and A. Ponse. *Kleene's Three-Valued Logic and Process Algebra.*

[P9722] J.A. Bergstra and A. Ponse. *Bochvar-McCarthy Logic and Process Algebra.*

[P9721] M.P.A. Sellink and C. Verhoef. *Reflections on the Evolution of COBOL.*

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

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[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.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

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

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

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF*'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

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

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for μCRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*

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

[P9419] J.C.M. Baeten, J.A. Bergstra, and Gh. Stefanescu. *Process algebra with feedback.*

[P9418] L.H. Oei. *Pruning the search tree of interlocking design and application language operational semantics.*

[P9417] B. Diertens. *New features in PSF I: interrupts, disrupts, and priorities.*

[P9416] S.M. Üsküdarlı. *Generating visual editors for formally specified languages.*

[P9415] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. *Control and data transfer in the distributed editor of the ASF+SDF meta-environment.*

[P9414] M.G.J. van den Brand and C. Groza. *The algebraic specification of annotated abstract syntax trees.*

[P9413] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen (editors). *Workshop on Algebra of Communicating Processes May 16-17, 1994 Utrecht University.*

[P9218b] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing probabilistic processes: ACP with generative probabilities (revised version of P9218).*

[P9412] C. Groza. *An experiment in implementing process algebra specifications in a procedural language.*

[P9411] M.J. Koens and L.H. Oei. *A real time muCRL specification of a system for traffic regulation at signalized intersections.*

[P9410] J.J. van Wamel. *Inductive proofs with sets, and some applications in process algebra.*

[P9409] J.F. Groote and J.J. van Wamel. *Algebraic data types and induction in muCRL.*

[P9408] J.A. Bergstra and P. Klint. *The toolbus: a component interconnection architecture.*

[P9407] P.H. Rodenburg. *Module algebra for initial algebra semantics.*

[P9406] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. *A propositional logic with 4 values: true, false, divergent and meaningless.*

[P9211b] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in real time process algebra (revised version of P9211).*

[P9405] J.C.M. Baeten and J.A. Bergstra. *Process algebra with partial choice.*

[P9314b] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with iteration and nesting (revised version of P9314).*

[P9404] J. van den Brink and W.O.D. Griffioen. *Formal semantics of discrete absolute timed interworkings.*

[P9403] J.A. Bergstra and Gh. Stefanescu. *Processes with multiple entries and exits modulo isomorphism and modulo bisimulation.*

[P9402] J.A. Bergstra and Ch. Stefanescu. *Bisimulation is two-way simulation.*

[P9401] J.C.M. Baeten and J.A. Bergstra. *Graph isomorphism models for non interleaving process algebra.*

[P9331] J.A. Hillebrand. *An algebraic specification of a manufacturing system with hierarchical control.*

[P9330] K.M.M. de Leeuw and H. van der Meer. *A turning grille from the ancestral castle of the Dutch Stadtholders.*

[P9329] J.J. Brunekreef. *Process specification in a UNITY format.*

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

[P9327] M.G.J. van den Brand. *Generation of language independent prettyprinters.*

[P9326] J.C.M. Baeten and J.A. Bergstra. *Non interleaving process algebra.*

[P9306b] J.A. Bergstra, A. Ponse, and J.J. van Wamel. *Process algebra with backtracking (revised version of P9306).*

[P9325] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra with infinitesimals.*

[P9324] J.J. Brunekreef, R.L.C. Koymans J.P. Katoen, and S. Mauw. *Design and analysis of dynamic leader election protocols in broadcast networks.*

[P9323] M. Kaart and I. Polak. *Het alternating bit protocol met time out in discrete tijd.*

[P9322] N.J. Drost. *Unification in an algebra with choice and sequential composition.*

[P9321] N.J. Drost. *Unification in an algebra with choice and action prefix.*

[P9320] K.M.M. de Leeuw and H. van der Meer. *Een roostergeheimschrift van Alexander baron van Spaen.*

[P9319] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with combinators.*

[P9318] I. Bethke and A. Ponse. *A car registration authority, a concise PSF-specification.*

[P9317] S.F.M. van Vlijmen and J.J. van Wamel. *A semantic approach to Protocold using process algebra.*

[P9316] J.J. Brunekreef and A. Ponse. *An algebraic specification of a model factory, part IV.*

[P9315] M.G.J. van den Brand. *Prettyprinting without losing comments.*

[P9314] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with iteration.*

[P9313] S.F.M. van Vlijmen and A. van Waveren. *Algebraic specification of a system for traffic regulation at signalized intersections.*

[P9312] J.W.C. Koorn and H.C.N. Bakker. *Building an editor from existing components: an exercise in software re-use.*

[P9311] I. Polak. *Specification of a bank account in PSF.*

[P9305b] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix (revised version of P9305).*

[P9310] A.S. Klusener, S.F.M. van Vlijmen, and A. van Waveren. *Service independent building blocks-I; concepts, examples and formal specifications.*

[P9309] J.C.M. Baeten, J.A. Bergstra, and R.N. Bol. *A real time process logic.*

[P9308] J.F.A.K. van Benthem and J.A. Bergstra. *Logic of transition systems.*

[P9307] A. Stins and A. Schoneveld. *Specification of a bank account with process algebra.*

[P9306] J.A. Bergstra, A. Ponse, and J.J. van Wamel. *Process algebra with backtracking.*

[P9305] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix.*

[P9304] S.F.M. van Vlijmen and A. van Waveren. *On generating synchronous interworkings from PSF process traces.*

[P9303] A. Ponse and J.A. Verschuren. *An algebraic specification of a model factory, part III.*

[P9302] J.A. Bergstra and J.V. Tucker. *The data type variety of stack algebras.*

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

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

[P9222] J.W.C. Koorn. *Connecting semantic tools to a syntax-directed user-interface.*

[P9221] J. Blanco. *Definability with the state operator in process algebra.*

[P9220] A. van Waveren. *Specification of remote sensing mechanisms in real space process algebra.*

[P9219] I. Bethke and P.H. Rodenburg. *Equational Constructor Induction.*

[P9218] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing Probabilistic Processes: ACP with Generative Probabilities.*

[P9217] A. Ponse. *Process algebra and dynamic logic.*

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

[P9216] E.E. Polak. *An efficient implementation of branching bisimulation and distinguishing formulae.*

[P9215] J.A. Bergstra and J.V. Tucker. *Equational specifications, complete term rewriting systems, and computable and semicomputable algebras.*

[P9214] J. A. Hillebrand and A. Ponse. *An algebraic specification of a model factory, part II.*

[P9213] N.J. Drost. *Unification in the algebra of sets with union and empty set.*

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

[P9211] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in realtime process algebra.*

[P9210] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. *Decidability of bisimulation equivalence for processes generating context-free languages.*

[P9209] S.F.M. van Vlijmen and A. van Waveren. *An algebraic specification of a model factory.*

[P9208] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra.*

[P9207] K. de Leeuw and H. van der Meer. *A secret message in the archive of the last great pensionary of Holland.*

[P9206] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

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

[P9204] C. Verhoef. *On induction principles.*

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

[P9202] J.W.C. Koorn. *A generic text and structure editor.*

[P9201] P.H. Rodenburg. *Interpolation in Equational Logic.*

[P9102b] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols (revised version of P9102).*

[P9112] A. Mateescu. *Recursive systems with auxiliary variables in Basic Process Algebra.*

[P9111] J.C.M. Baeten and J.A. Bergstra. *A survey of axiom systems for process algebras.*

[P9110] J.J. van Wamel. *A formal specification of three simple protocols.*

[P9109] J.J. Brunekreef. *A formal specification of two simple protocols for Local Area Networks.*

[P9108] J.J. Brunekreef. *A formal specification of the Amoeba Transaction Protocol.*

[P9107] G.J. Veltink. *The PSF toolkit.*

[P9106] J.C.M. Baeten and J.A. Bergstra. *Asynchronous communication in real space process algebra.*

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

[P9104] J.C.M. Baeten and J.A. Bergstra. *The state operator in real time process algebra.*

[P9103] H.P. Korver. *Computing distinguishing formulas for branching bisimulation.*

[P9102] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols.*

[P9101] H.P. Korver. *The current state of bisimulation tools.*

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

[P9008] J.C.M. Baeten and J.A. Bergstra. *Process algebra with signals and conditions.*

[P9007] S. Mauw and Gy. Max. *A formal specification of the Ethernet protocol.*

[P9006] G.J. Akkerman and J.C.M. Baeten. *Term rewriting analysis in process algebra.*

[P9005] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9004] N.J. Drost. *Algbraic formulations of trace theory.*

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

[P8804b] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions (revised version of P8804).*

[P8910b] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (revised version of P8910).*

[P8906b] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD (revised version of P8906).*

[P9003] C. Verhoef. *On the register operator.*

[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.*

[P9001] L.W. Kuurman. *The jungle of process semantics.*

[P8916] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra.*

[P8915] J.C. Mulder. *The inevitable coffee machine.*

[P8914] H. Jacobsson and S. Mauw. *A Token ring network in PSFd.*

[P8913] H.R. Walters. *Hybrid implementations of algebraic specifications.*

[P8912] S. Mauw and G.J. Veltink. *A Tool Interface Language for PSF.*

[P8911] P.H. Rodenburg. *Interpolation in conditional equational logic.*

[P8910] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (preliminary version).*

[P8909] J.A. Bergstra. *Algebra of states and transitions.*

[P8908] S. Mauw and F. Wiedijk. *Specification of the transit node in PSFd.*

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

[P8907] J.A. Bergstra. *Kerninformatica en toekomst.*

[P8906] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD.*

[P8815b] J.A. Bergstra. *Process algebra for synchronous communication and observation (revised version of P8815).*

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

[P8904] J.A. Bergstra, S. Mauw, and F. Wiedijk. *Uniform algebraic specifications of finite sorts with equality.*

[P8903] J.A. Bergstra. *A representation of addition and deletion lists using module algebra.*

[P8902] R.P. Ogilvie. *Knowledge engineering vs. software engineering.*

[P8901] S. Mauw and G.J. Veltink. *An introduction to PSF.*

[P8826] J. Treur. *Design of modular and interactive knowledge-based systems.*

[P8825] H.W. Lenferink. *Local control of inference by means of meta-rules.*

[P8824] J.A. Bergstra and J.W. Klop. *Process theory based on bisimulation semantics.*

[P8823] J.A. Bergstra, J. Heering, and P. Klint. *Module Algebra (revised version).*

[P8822] J.L.M. Vrancken. *The algebraic specification of semicomputable data types (revised version).*

[P8821] J.C.M. Baeten and F.W. Vaandrager. *Specification and verification of a circuit in ACP (revised version).*

[P8820] J. Treur. *A logical framework for design processes.*

[P8819] H.K. Faber. *Strategische uitleg in een medisch expertsysteem.*

[P8818] H.K. Faber. *Uitleg en kennisrepresentatie in expertsystemen.*

[P8817] J. Treur. *Heuristic reasoning with incomplete knowledge.*

[P8816] J.A. Bergstra. *ACP with signals.*

[P8815] J.A. Bergstra. *Process algebra for synchronous communication and observation.*

[P8814] S. Mauw and G.J. Veltink. *A process specification formalism.*

[P8813] J. Treur. *Reasoning about partial models, actions and plans.*

[P8812] J. Treur. *On the use of reflection principles in modelling complex reasoning.*

[P8811] F. Wiedijk. *Voorlopig rapport over de specificatie-taal Perspect.*

[P8810] R.E. Swart. *The BCA Bull Course Adviser.*

[P8809] J. Treur. *Metakennis en meta-inferenties in expertsystemen.*

[P8808] J.A. Bergstra. *A mode transfer operator in process algebra.*

[P8807] J.L.M. Vrancken. *The implementation of process algebra specifications in POOL-T.*

[P8806] J. Treur. *Generic inference processes and their interactions in complex diagnostic tasks; a logical description.*

[P8805] J. Treur. *Completeness and definability in diagnostic expert systems.*

[P8804] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions.*

[P8803] J.C.M. Baeten and F.W. Vaandrager. *Specification and verification of a circuit in ACP.*

[P8802] J.C.M. Baeten and J.A. Bergstra. *Recursive process definitions with the state operator.*

[P8801] F.R. Burggraaff and E. van der Meulen. *ASF Specification of a B-tree of order 1.*

[P8714] L.G. Bouma and H.R. Walters. *Implementing algebraic specifications.*

[P8713] W. Syski. *On a certain probabilistic approximation method for reasoning with uncertainty.*

[P8712] W. Syski and J. Treur. *Reasoning about uncertainty represented by meta-reasoning, the endorsements approach.*

[P8711] J. Treur. *Een logische analyse van diagnostische redeneerprocessen; redeneren met en over hypothesen.*

[P8710] J. Treur. *Volledigheid en definieerbaarheid in diagnostische redeneersystemen.*

[P8709] J.C.M. Baeten and J.A. Bergstra. *Global renaming operators in concrete process algebra (revised version).*

[P8708] S. Mauw. *Process algebra as a tool for the specification and verification of CIM-architectures.*

[P8707] J.H. Verhagen. *Expertsystemen bij de Nederlandse Spoorwegen.*

[P8706] J.C.M. Baeten, J.A. Bergstra, and J.L.M. Vrancken. *Processen en procesexpressies.*

[P8705] J.L.M. Vrancken. *The algebraic specification of semicomputable datatypes.*

[P8704] F. Wiedijk. *Termherschrijfsystemen in Prolog.*

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

[P8702] M.R. Dasselaar. *Development of an expert system, from theory to practice.*

[P8701] R.A. Groenveld. *Verification of a sliding window protocol by means of process algebra.*