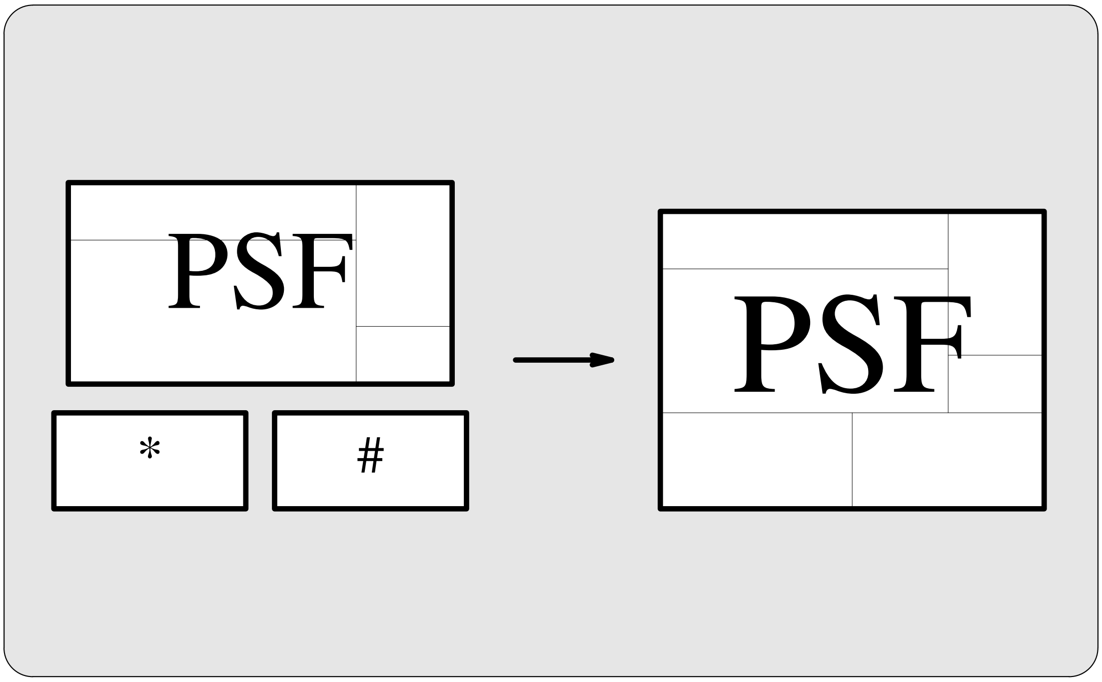
University of Amsterdam
Programming Research Group



New Features in PSF II

Iteration and Nesting

B. Diertens
A. Ponse

Report P9425 October 1994

University of Amsterdam
Department of Mathematics and Computer Science
Programming Research Group

New features in PSF II
iteration and nesting

B. Diertens
A. Ponse

Report P9425 October 1994

B. Diertens

Programming Research Group
Faculty of Mathematics and Computer Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7554
e-mail: bobd@fwi.uva.nl

A. Ponse

Programming Research Group
Faculty of Mathematics and Computer Science
University of Amsterdam

Kruislaan 403
1098 SJ   Amsterdam
The Netherlands

tel. +31 20 525.7592
e-mail: alban@fwi.uva.nl

This document (including the cover) is completely formatted using the Groff document formatting system, which is copyrighted and committed to the world by the Free Software Foundation, Inc.

Cover: Bob Diertens

Universiteit van Amsterdam, 1994

1. Introduction

PSF has already been provided with interrupts, disrupts, and priorities (see [Die94]). Now, PSF has been extended with an iteration and a nesting operator.

The description of the extension of PSF can be found in chapter 2. The adjustments of the toolkit are described in chapter 3. In chapter 4, an example is given on how to use the new additions. The remainder of this chapter gives a short description of PSF and the toolkit developed up till now.

1.1 PSF

PSF (Process Specification Formalism) is a Formal Description Technique developed for the specification of concurrent systems. A description of PSF can be found in [MauVel90], [MauVel93], and [Die94].

PSF has been designed as the base for a set of tools to support ACP (Algebra of Communicating Processes) [BerKlo86]. It is very close to the informal syntax normally used in denoting ACP-expressions. The part of PSF that deals with the description of data is based on ASF (Algebraic Specification Formalism) [BerHeeKli89]. To meet the modern needs of software engineering, PSF supports the modular construction of specifications and the parametrization of modules.

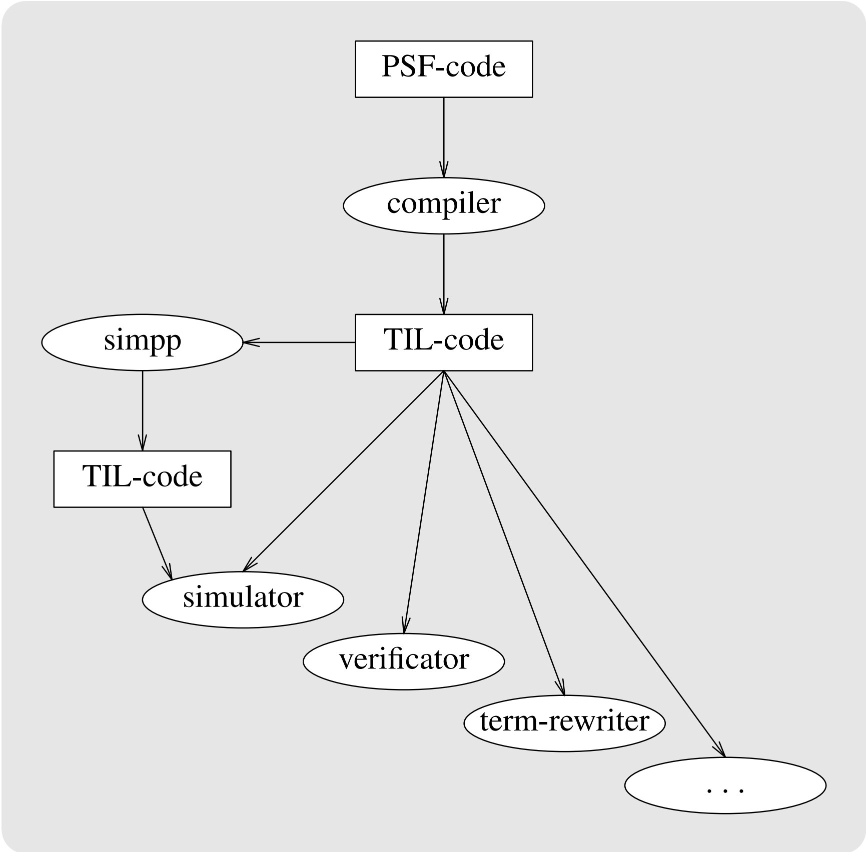
Processes in PSF are described as a series of atomic actions combined by operators. Atomic actions are the basic and indivisible elements of processes in PSF. By using atomic actions and operators we can construct *process expressions*. These process expressions in combination with recursive process definitions are used to define processes. The basic operators on processes are sequential, alternative and parallel composition.

Communication between parallel processes can be defined with the communication function, which takes two atoms as arguments and returns the result of their communication. The *encapsulation* operator can be used to rename a set of atomic actions into *delta*, the constant process indicating a deadlock. This is used to enforce communication between parallel processes. The *hiding* operator can be used to rename a set of atomic actions into *skip*, the constant process indicating an internal action. This operator makes it possible to concentrate on a set of visible actions.

The language has also been provided with an *interrupt*,*disrupt*, and *priority* mechanism.

1.2 The PSF-Toolkit

At the center of the Toolkit is the Tool Interface Language (TIL), through which all tools can communicate.



The PSF-compiler has three main stages, a parser, a normalizer, and a translator. It makes use of a library-manager to support and control the separate compilation of PSF-modules. It is also possible to make use of a standard library.

The Simulator shows traces of selected items, when it simulates a specification. It is possible to set breakpoints on atoms and processes. The user can choose the actions to perform from a list, but simulation can also be done randomly. The Simulator can only handle *sums* and *merges* over sets that consist of an enumeration. The tool simpp (simulator preprocessor) can be used to try to make an enumerated set out of a sort or set, to overcome this problem.

The Verificator makes it possible to manipulate process-expressions in an axiomatic way. It has been provided with several algorithms, which take over the tiresome job of applying a lot of axioms to a term. The resulting proof can be written to a file.

The Term-rewriter is a standard term rewriting program, that supports conditional equations using the rightmost innermost rewriting strategy. The kernel of the Term-rewriter is used by other programs in the Toolkit.

2. Description of the extensions

The new features, iteration and nesting, are described in [BerBetPon94]. Here, * is used as an infix-operator for iteration and # for nesting. The first is also referred to as the Kleene star, since he was the first who used it in a similar construction (see [Kle56]).

In process algebra, recursion is used as a means to specify infinite behaviour. The purpose of the *- and #-operators is to make it possible to specify this without recursion.

2.1 Iteration

The *-operator is called BKS (Binary Kleene Star). The expression $x * y$ is the process that chooses between x and y, and upon termination of x has this choice again. For example, the process $a * b$ for atoms a and b, can be depicted by

$$\begin{tikzcd} a * b \arrow[r, "b"] \arrow[loop left, "a"] & \surd \end{tikzcd}$$

where $\surd$ is a symbol expressing (successful) termination.

The algebraic laws and the transition rules are given in the following tables.

BKS1	$x \cdot (x * y) + y$	$= x * y$
BKS2	$x * (y \cdot z)$	$= (x * y) \cdot z$
BKS3	$x * (y \cdot ((x + y) * z) + z)$	$= (x + y) * z$
BKS4	$\partial_H(x * y)$	$= \partial_H(x) * \partial_H(y)$
BKS5	$\tau_I(x * y)$	$= \tau_I(x) * \tau_I(y)$

TABLE 2-1. Algebraic laws for *

bks1	$\dfrac{x \xrightarrow{a} x'}{x * y \xrightarrow{a} x' \cdot (x * y)}$
bks2	$\dfrac{x \xrightarrow{a} \surd}{x * y \xrightarrow{a} x * y}$
bks3	$\dfrac{y \xrightarrow{a} y'}{x * y \xrightarrow{a} y'}$
bks4	$\dfrac{y \xrightarrow{a} \surd}{x * y \xrightarrow{a} \surd}$

TABLE 2-2. Transition rules for $*$

2.2 Nesting

The #-operator is called NO (Nesting Operator). The expression $x \# y$ is the process that chooses between x and y, and upon termination of x has the choice to perform x again, or to perform y and upon termination of y repeat the number of times x has already be performed. For example, the process $a \# b$ for atoms a and b, can be illustrated by

$$
\begin{tikzcd}
a \# b \arrow[r, "b"] \arrow[d, "a"'] & \surd \\
(a \# b) \cdot a \arrow[r, "b"] \arrow[d, "a"'] & a \arrow[u, "a"'] \\
(a \# b) \cdot a^2 \arrow[r, "b"] \arrow[d, "a"'] & a^2 \arrow[u, "a"'] \\
(a \# b) \cdot a^3 \arrow[r, "b"] & a^3 \arrow[u, "a"'] \\
\vdots &
\end{tikzcd}
$$

The algebraic laws and the transition rules are given in the following tables.

NO1	$x \cdot (x \# y) \cdot x + y$	$= x \# y$
NO2	$(x \# y) \cdot x$	$= x \# (y \cdot x)$
NO3	$(x + y) \# (x \cdot ((x + y) \# z) \cdot (x + y) + z)$	$= (x + y) \# z$
NO4	$\partial_H(x \# y)$	$= \partial_H(x) \# \partial_H(y)$
NO5	$\tau_I(x \# y)$	$= \tau_I(x) \# \tau_I(y)$
NO6	$x \# \delta$	$= x * \delta$
NO7	$\tau \# x$	$= \tau * x$

TABLE 2-3. Algebraic laws for #

no1	$\dfrac{x \xrightarrow{a} x'}{x \# y \xrightarrow{a} x' \cdot ((x \# y) \cdot x)}$
no2	$\dfrac{x \xrightarrow{a} \surd}{x \# y \xrightarrow{a} (x \# y) \cdot x}$
no3	$\dfrac{y \xrightarrow{a} y'}{x \# y \xrightarrow{a} y'}$
no4	$\dfrac{y \xrightarrow{a} \surd}{x \# y \xrightarrow{a} \surd}$

TABLE 2-4. Transition rules for #

3. Adjusting the PSF-Toolkit

This chapter describes the adjustments made to the several tools in the toolkit in order to cope with the newly added features of PSF.

3.1 The compiler

Since PSF already has binary operators, it is easy to add two more such operators. We only have to deal with the priorities of the new operators in the parser. But we use a parser generator which has the possibility to add priorities easily. So, there are no complications here.

3.2 TIL

Of course, the Tool Interface Language had to be extended with the *- and #-operators. These extensions are only of the form of two names for these operators, and there were no new constructions necessary.

3.3 The simulator

The mechanism to implement the *- and #-operators, were already there in the simulator. We only had to combine them in the right way.

The *-operator is simulated by making a process-tree for both operands. If an atom from the process-tree of the first argument is executed, the other process-tree is disabled. On completion of the execution of the first operand, the other process-tree is enabled and a new process-tree for the first operand is made.

If an atom from the process-tree of the second argument is executed, the tree for the first argument is removed.

The #-operator is simulated in the same way, except for that a counter is added to record the number of times the first argument is executed. On completion of the execution of the second argument, a new process-tree for the first argument is made. And, everytime the execution of this tree finishes, the counter is decremented and if it is still greater than zero, a new process-tree for the first argument is made.

3.3.1 The verificator

The verificator and simulator have a similar structure for process-control, so adjusting the verificator can be done in a similar manner.

4. Examples

As an example, we show how to specify a stack, which originates from [BerBetPon94]. For simplicity, we take the non-terminating variant with dataset $D = \{d_1, d_2\}$, so we have

$$S_\in = \sum_j r(d_j) \cdot S_{d_j}$$

$$S_{dw} = \sum_j r(d_j) \cdot S_{d_j dw} + s(d) \cdot S_w$$

$S_\in$ represents the empty stack, and the contents of a non-empty stack is represented by the index of S: S_{dw} is the stack that contains dw with d on top. An action $r(d_i)$ (receive d_i) models the push of d_i onto the stack, and an action $s(d_i)$ (send d_i) models a pop of d_i from the stack.

We have two "half-counters" represented by:

$$HC_j = ((\bar{a}_j \# \bar{b}_j) \cdot \bar{c}_j) * \delta \quad (j = 1, 2)$$

Let γ for $j = 1, 2$ be defined such that

$$\gamma(a_j, \bar{a}_j) = \gamma(b_j, \bar{b}_j) = \gamma(c_j, \bar{c}_j) = i$$

Defining

$$H = \{a_j, \bar{a}_j, b_j, \bar{b}_j, c_j, \bar{c}_j \mid j = 1, 2\}$$

$$I = \{i\}$$

we can specify P_1 such that

$$\tau_I \circ \partial_H(P_1 \parallel HC_1 \parallel HC_2)$$

behaves as the stack $S_\in$ (cf. Theorem 4.2 in [BerBetPon94]).

In section 4.1 we specify the stack by using the *-operator, resulting in a stack with * and #. We give a PSF representation of this.

4.1 Representation of the Stack with * and

We recall

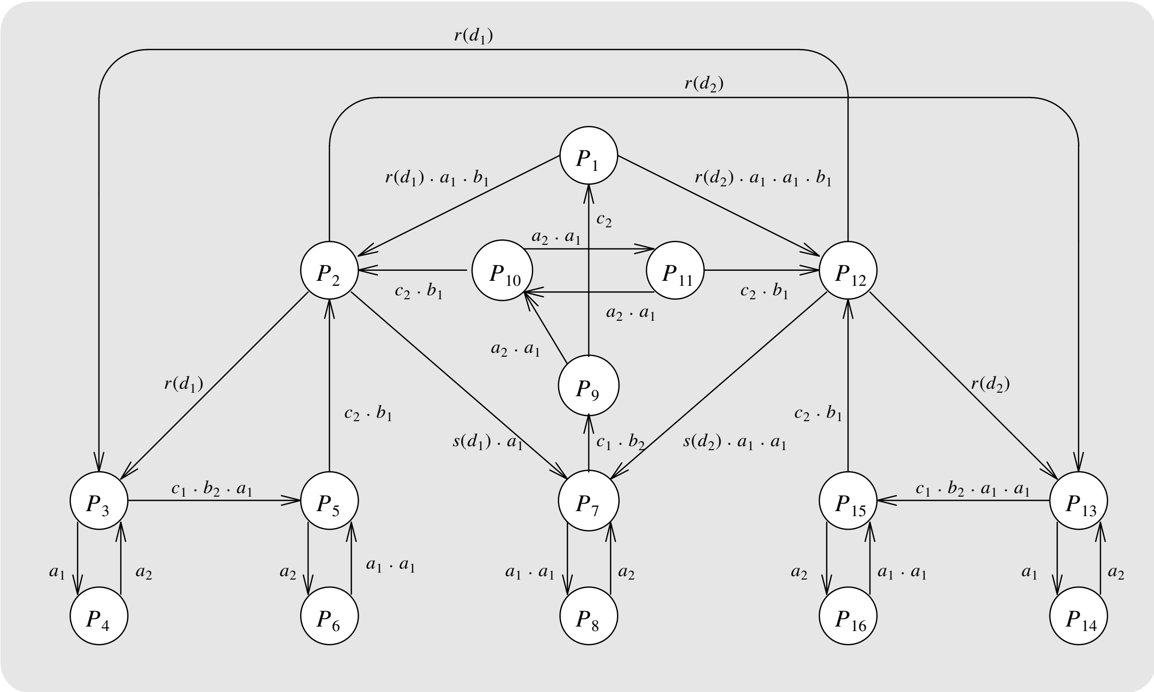
Theorem 4-1 (Theorem 3.4 in [BerBetPon94]). *For each regular process P over A_δ there is a finite extension B of A such that P can be expressed in $ACP^*_\tau(B, \gamma)$ with handshaking only, and the actions in A not subject to communication.*

This proof starts from a linear specification of the form

$$U_i = \sum_{j=1}^{n} (\alpha_{i,j} \cdot U_j) + \beta_i \tag{1}$$

where $\alpha_{i,j}$ and β_i are finite sums of actions or δ, and the $a_{j,j} = \delta$ for all $j \in \{1, \ldots, n\}$. Each regular process can indeed be characterized by such a linear form.

We shall employ a trivial modification of the construction in the proof of Theorem 3.4 in [BerBetPon94]: we assume that the $\alpha_{i,j}$ and β_i are finite sums of either actions or δ, or *finite products* of actions. The proof strategy of [BerBetPon94] still applies in this case, in particular to the process P_1 as depicted below.



Or, as a recursive specification:

$$P_i = \sum_{j=1}^{16} \alpha_{i,j} \cdot P_j \quad \text{for } i = 1, \ldots, 16$$

where all $\alpha_{i,j}$ equal δ, except for

$$\begin{array}{llll}
\alpha_{1,2} & = r(d_1) \cdot a_1 \cdot b_1 & \alpha_{1,12} & = r(d_2) \cdot a_1 \cdot a_1 \cdot b_1 \\
 & & & \\
\alpha_{2,3} & = r(d_1) & \alpha_{12,13} & = r(d_2) \\
\alpha_{2,7} & = s(d_1) \cdot a_1 & \alpha_{12,7} & = s(d_2) \cdot a_1 \cdot a_1 \\
\alpha_{2,13} & = r(d_2) & \alpha_{12,3} & = r(d_1) \\
\alpha_{3,4} & = a_1 & \alpha_{13,14} & = a_1 \\
\alpha_{3,5} & = c_1 \cdot b_2 \cdot a_1 & \alpha_{13,15} & = c_1 \cdot b_2 \cdot a_1 \cdot a_1 \\
\alpha_{4,3} & = a_2 & \alpha_{14,13} & = a_2 \\
\alpha_{5,2} & = c_2 \cdot b_1 & \alpha_{15,12} & = c_2 \cdot b_1 \\
\alpha_{5,6} & = a_2 & \alpha_{15,16} & = a_2 \\
\alpha_{6,5} & = a_1 \cdot a_1 & \alpha_{16,15} & = a_1 \cdot a_1 \\
 & & & \\
\alpha_{7,8} & = a_1 \cdot a_1 & \alpha_{7,9} & = c_1 \cdot b_2 \\
\alpha_{8,7} & = a_2 & &
\end{array}$$

$$\begin{array}{llll} \alpha_{9,1} & = c_2 & \alpha_{9,10} & = a_2 \cdot a_1 \\ \alpha_{10,2} & = c_2 \cdot b_1 & \alpha_{10,11} & = a_2 \cdot a_1 \\ \alpha_{11,10} & = a_2 \cdot a_1 & \alpha_{11,12} & = c_2 \cdot b_1 \end{array}$$

Next we give a representation of P_1 in $ACP^*_\tau(B, \gamma)$. We take

$$B \stackrel{def}{=} A \cup H \cup I \cup \bigcup_{j=1}^{16} (H_j \cup I_j)$$

where

$$\begin{array}{ll} H & \stackrel{def}{=} \{r_i, s_i \mid i \in \{1, \ldots, 16\}\} \\ I & \stackrel{def}{=} \{t_i \mid i \in \{1, \ldots, 16\}\} \\ H_j & \stackrel{def}{=} \{k_j, l_j, m_j\} \\ I_j & \stackrel{def}{=} \{i_j\} \end{array}$$

The only communications defined are:

$$r_j \mid s_j = t_j$$
$$k_j \mid k_j = i_j$$
$$l_j \mid m_j = i_j$$

for $j \in \{1, \ldots, 16\}$.

Let $j, k \in \{1, \ldots, 16\}$. From [BerBetPon94] it follows that

$$P_1 = \tau_I \circ \partial_H \left(T'_1 \parallel (\parallel_{i \in \{2, \ldots, 16\}} T_i) \right)$$

where the T'_j are defined by

$$T'_j = \sum_{k=1}^{16} (\alpha_{j,k} \cdot s_k \cdot T_j)$$

and where T_j is defined by

$$T_j = \tau_{I_j} \circ \partial_{H_j} \left(r_j \cdot \left(\sum_k (\alpha_{j,k} \cdot s_k) \right) \cdot Q_j \parallel S_j \right)$$

$$Q_j = \left(k_j \cdot r_j \cdot \left(\sum_k (\alpha_{j,k} \cdot s_k) \right) \right) * m_j$$

$$S_j = k_j * (m_j \cdot l_j)$$

So the Q_j and S_j are the only *-processes involved in the specification of P_1.

4.1.1 The stack in PSF

First we specify some numbers to be used as indices.

```
data module Numbers
begin
   exports
   begin
      sorts
         N
      functions
         1     : -> N
         2     : -> N
         3     : -> N
         4     : -> N
         5     : -> N
         6     : -> N
         7     : -> N
         8     : -> N
         9     : -> N
         10    : -> N
         11    : -> N
         12    : -> N
         13    : -> N
         14    : -> N
         15    : -> N
         16    : -> N
   end
end Numbers
```

And the data we use

```
data module Data
begin
   exports
   begin
      sorts
         Data
      functions
         d1    : -> Data
         d2    : -> Data
   end
end Data
```

Since it is not possible to specify sets in a data module and sets must be declared before used, we specify them in a special process module.

```
process module Sets
begin
   exports
   begin
      sets
         of N
            P-index = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 } -- indices for processes
            A-index = { 1, 2 } -- indices for actions
   end
   imports
      Numbers
end Sets
```

Now, we can specify the stack.

```
process module Stack
begin
  imports
    Sets, Data
  atoms
    r, s          : Data
    a, b, c       : A-index
    a', b', c'    : A-index
    r, s, t       : P-index
    k, l, m       : P-index
    i
    i             : P-index
  processes
    Stack
    HC     : A-index
    X
    T      : P-index
    T'     : P-index
    Q      : P-index
    S      : P-index
    Alpha  : P-index # P-index
  sets
    of P-index
      P-index'   = P-index \ { 1 }
    of atoms
      H-j  = { k(j), l(j), m(j) | j in P-index }
      I-j  = { i(j) | j in P-index }
      XH   = { r(i), s(i) | i in P-index }
      XI   = { t(i) | i in P-index }
      H    = { a(j), a'(j), b(j), b'(j), c(j), c'(j) | j in A-index }
      I    = { i }
  communications
    a(j) | a'(j)    = i          for j in A-index
    b(j) | b'(j)    = i          for j in A-index
    c(j) | c'(j)    = i          for j in A-index
    r(j) | s(j)     = t(j)       for j in P-index
    k(j) | k(j)     = i(j)       for j in P-index
    l(j) | m(j)     = i(j)       for j in P-index
  variables
    x      : -> A-index
    j      : -> P-index
  definitions
    Stack  = hide(I, encaps(H, X || HC(1) || HC(2)))

    HC(x)  = ((a'(x) # b'(x)) . c'(x)) * delta

    X      = hide(XI, encaps(XH, T'(1) || merge(j in P-index', T(j))))
    T'(j)  = sum(k in P-index, Alpha(j, k) . s(k) . T(j))
    T(j)   = hide(I-j, encaps(H-j, (r(j) . sum(k in P-index, Alpha(j, k) . s(k))) . (Q(j) || S(j))))
    Q(j)   = (k(j) . r(j) . sum(k in P-index, Alpha(j, k) . s(k))) * m(j)
    S(j)   = k(j) * (m(j) . l(j))

    Alpha(1, 2)       = r(d1) . a(1) . b(1)
    Alpha(1, 12)      = r(d2) . a(1) . a(1) . b(1)

    Alpha(2, 3)       = r(d1)
```

```
    Alpha(2, 7)      = s(d1) . a(1)
    Alpha(2, 13)     = r(d2)
    Alpha(3, 4)      = a(1)
    Alpha(3, 5)      = c(1) . b(2) . a(1)
    Alpha(4, 3)      = a(2)
    Alpha(5, 2)      = c(2) . b(1)
    Alpha(5, 6)      = a(2)
    Alpha(6, 5)      = a(1) . a(1)

    Alpha(12, 13)    = r(d2)
    Alpha(12, 7)     = s(d2) . a(1) . a(1)
    Alpha(12, 3)     = r(d1)
    Alpha(13, 14)    = a(1)
    Alpha(13, 15)    = c(1) . b(2) . a(1) . a(1)
    Alpha(14, 13)    = a(2)
    Alpha(15, 12)    = c(2) . b(1)
    Alpha(15, 16)    = a(2)
    Alpha(16, 15)    = a(1) . a(1)

    Alpha(7, 8)      = a(1) . a(1)
    Alpha(8, 7)      = a(2)
    Alpha(9, 1)      = c(2)
    Alpha(10, 2)     = c(2) . b(1)
    Alpha(11, 10)    = a(2) . a(1)

    Alpha(7, 9)      = c(1) . b(2)
    Alpha(9, 10)     = a(2) . a(1)
    Alpha(10, 11)    = a(2) . a(1)
    Alpha(11, 12)    = c(2) . b(1)
end Stack
```

Note that we used *H-j* and *I-j* instead of $H(j)$ and $I(j)$. We cannot specify that sort of sets, so we enlarged the sets with all the possible indices.
We also used a', b', and c' instead of $\bar{a}$, $\bar{b}$, and $\bar{c}$ and *Alpha* instead of α.

5. References

[BerBetPon94] J.A. Bergstra, I. Bethke, and A. Ponse, "Process Algebra with Iteration and Nesting," *The Computer Journal*, vol. 37, no. 4, pp. 243-258, 1994.

[BerHeeKli89] J.A. Bergstra, J. Heering, and P. Klint, "The Algebraic Specification Formalism ASF," in *Algebraic Specification*, ed. J.A. Bergstra, J. Heering, P. Klint, ACM Press Frontier Series, pp. 1-66, Addison-Wesley, 1989.

[BerKlo86] J.A. Bergstra and J.W. Klop, "Process algebra: specification and verification in bisimulation semantics," in *Math. & Comp. Sci. II*, ed. M. Hazewinkel, J.K. Lenstra, L.G.L.T. Meertens, eds., CWI Monograph 4, pp. 61-94, North-Holland, Amsterdam, 1986.

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

[Kle56] S.C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata studies*, pp. 3-41, Princeton University Press, 1956.

[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.