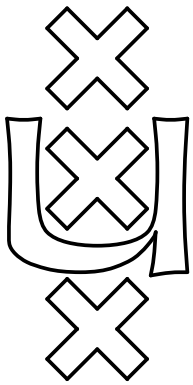
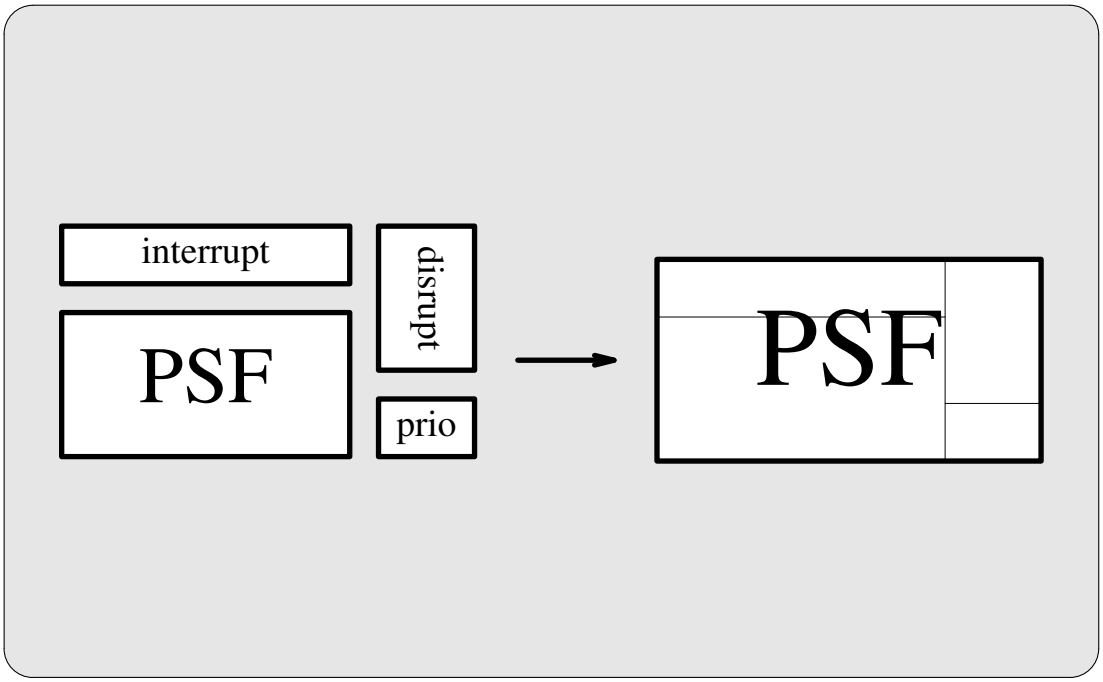


University of Amsterdam

Programming Research Group

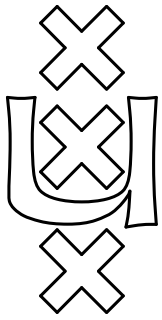


New Features in PSF I

Interrupts, Disrupts, and Priorities

B. Diertens

Report P9417 June 1994



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

New features in PSF I

interrupts, disrupts, and priorities

B. Diertens

Report P9417 June 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

Cover: Bob Diertens

Universiteit van Amsterdam, 1994

1. Introduction

PSF in its original form, only supports the basic features of ACP. There have been numerous proposals to extend ACP, in order to make it possible to specify in a more concise and realistic manner.

To facilitate specifying in PSF, the language has been extended with an interrupt and disrupt mechanism, and priorities. And so, the PSF-Toolkit has been adjusted to support the new features.

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] and [MauVel93].

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.

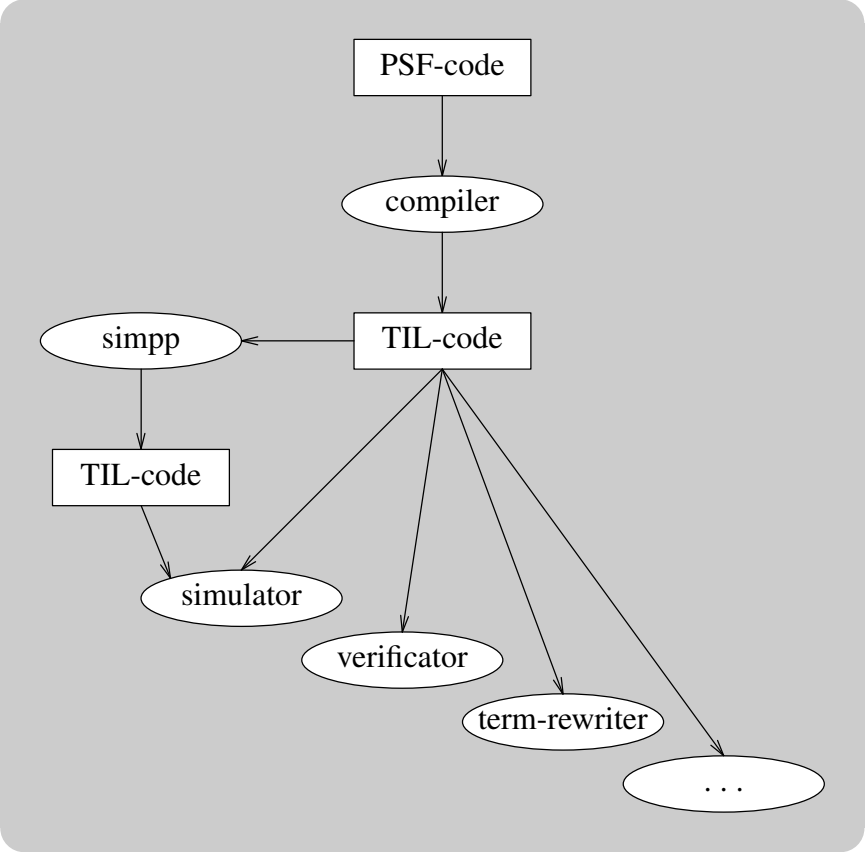
Processes in PSF are described as a series of atomic actions combined by operators. Atomic actions are the basic and indivisible elements of processes in PSF. By using atomic actions and operators we can construct *process expressions*. These process expressions in combination with recursive process 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.

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

2.1 Interrupts and Disrupts

The new features, interrupt and disrupt, are added to PSF in the form described in [Mau91]. The disrupt operator is identical to the mode transfer operator of [Ber89]. It should be noticed that LOTOS [ISO89] provides a disrupt mechanism as well.

The expression $interrupt(x, y)$ is used to express that process x can be interrupted at any time by process y. After y has finished, x resumes. The expression $disrupt(x, y)$ means that process x can be disrupted by y at any time. If y has finished, the whole process is finished. Both $interrupt(x, y)$ and $disrupt(x, y)$ finish if x finishes.

The algebraic laws for these operators are copied here. The x, y, and z denote process expressions, a, b, and c are atoms, and $\phi \rightarrow x$ denotes a guarded command.

INT	$interrupt(x, y)$	$= dinterrupt(x, y) + enable(y \,.\, interrupt(x, y), x)$
DINT1	$dinterrupt(a, x)$	$= a$
DINT2	$dinterrupt(a \,.\, x, y)$	$= a \,.\, interrupt(x, y)$
DINT3	$dinterrupt(x + y, z)$	$= dinterrupt(x, z) + dinterrupt(y, z)$
DINT4	$dinterrupt(\delta, x)$	$= \delta$
DINT5	$dinterrupt(\phi \rightarrow x, y)$	$= \phi \rightarrow dinterrupt(x, y)$
EN1	$enable(x, a)$	$= x$
EN2	$enable(x, y \,.\, z)$	$= enable(x, y)$
EN3	$enable(x, y + z)$	$= enable(x, y) + enable(x, z)$
EN4	$enable(x, \delta)$	$= \delta$
EN5	$enable(x, \phi \rightarrow y)$	$= \phi \rightarrow enable(x, y)$
DIS1	$disrupt(a, x)$	$= a + x$
DIS2	$disrupt(a \,.\, x, y)$	$= a \,.\, disrupt(x, y) + y$
DIS3	$disrupt(x + y, z)$	$= disrupt(x, z) + disrupt(y, z)$
DIS4	$disrupt(\delta, x)$	$= \delta$
DIS5	$disrupt(\phi \rightarrow x, y)$	$= \phi \rightarrow disrupt(x, y)$

TABLE 2-1. Algebraic laws for interrupt and disrupt

In the definition of the interrupt operator, the *dinterrupt* (delayed interrupt) operator is used. It behaves exactly like the interrupt operator, with the restriction that it cannot start with the interrupting process. The reason for introducing the delayed interrupt operator is

that setting

$$interrupt(x + y, y) = interrupt(x, z) + interrupt(y, z)$$

would result in

$$\begin{aligned} interrupt(a + b, c) &= interrupt(a, c) + interrupt(b, c) \\ &= a + c.\, interrupt(a, c) + b + c.\, interrupt(b, c) \end{aligned}$$

instead of

$$\begin{aligned} interrupt(a + b, c) &= dinterrupt(a + b, c) + c.\, interrupt(b, c) \\ &= a + b + c.\, interrupt(a + b, c) \end{aligned}$$

The first implies that the choice between a and b can be forced by executing one of the two possible c actions.

The second extra operator used here is *enable*, which denotes that the first argument can only be executed if the second one is not equal to deadlock.

The transition rules are given in the following table.

dis1	$\dfrac{x \xrightarrow{a} x'}{disrupt(x, y) \xrightarrow{a} disrupt(x', y)}$	int1	$\dfrac{x \xrightarrow{a} x'}{interrupt(x, y) \xrightarrow{a} interrupt(x', y)}$
dis2	$\dfrac{x \xrightarrow{a} \surd}{disrupt(x, y) \xrightarrow{a} \surd}$	int2	$\dfrac{x \xrightarrow{a} \surd}{interrupt(x, y) \xrightarrow{a} \surd}$
disr3	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} y'}{disrupt(x, y) \xrightarrow{b} y'}$	int3	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} y'}{interrupt(x, y) \xrightarrow{b} y'.\, interrupt(x, y)}$
dis4	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{disrupt(x, y) \xrightarrow{b} y'}$	int4	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{interrupt(x, y) \xrightarrow{b} y'.\, interrupt(x, y)}$
dis5	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} \surd}{disrupt(x, y) \xrightarrow{b} \surd}$	int5	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} \surd}{interrupt(x, y) \xrightarrow{b} interrupt(x, y)}$
dis6	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \surd}{disrupt(x, y) \xrightarrow{b} \surd}$	int6	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \surd}{interrupt(x, y) \xrightarrow{b} interrupt(x, y)}$

TABLE 2-2. Transition rules for interrupt and disrupt

2.2 Priorities

The interrupt and disrupt operators give the possibility to interrupt or disrupt a process, but these actions are not forced. In most cases, execution of a process may not continue when an interrupt or disrupt appears. So we need something to force this.

In [BaeBerKlo86] a priority operator is described. [Mau91] suggest a formalization of this operator. This formalization, however, is not strong enough for our needs. We will use the following operator:

$$prio(X_1 > X_2 > \cdots > X_n, process\text{-}expression)$$

with X_1, ..., X_n sets of atoms. This means that atoms in set X_1 have higher priority than

atoms in the sets X_2, ..., X_n, and atoms in set X_2 have higher priority than atoms in the sets X_3, ..., X_n, etc. If an atom is a member of more than one of these sets, the lower priority is ignored, so

$$prio(X > Y > Z, process\text{-}expression)$$

really means

$$prio(X > Y \backslash X > (Z \backslash Y) \backslash X, process\text{-}expression)$$

The form

$$prio(X, process\text{-}expression)$$

may also be used, and means that atoms in set X have higher priority than all other atoms in *process-expression*.

The algebraic laws for the prio operator are given in the following table.

PRI1	$prio(S > C, x)$	$= x \lhd_{S>C} \delta$	
PRI2	$prio(S, x)$	$= x \lhd_{S > atoms} \delta$	
PRI3	$a \lhd_S x$	$= a$	
PRI4	$a \lhd_{S>C} b$	$= a$	if $a \in S$
PRI5	$a \lhd_{S>C} b$	$= block(C, a)$	if $a \notin S \wedge b \in S$
PRI6	$a \lhd_{S>C} b$	$= a \lhd_C b$	if $a \notin S \wedge b \notin S$
PRI7	$a \lhd_{S>C} \delta$	$= a$	
PRI8	$\delta \lhd_{S>C} x$	$= \delta$	
PRI9	$a \lhd_{S>C} skip$	$= a$	
PRI10	$skip \lhd_{S>C} x$	$= skip$	
PRI11	$x \lhd_{S>C} y.\ z$	$= x \lhd_{S>C} y$	
PRI12	$x \lhd_{S>C} (y + z)$	$= (x \lhd_{S>C} y) \lhd_{S>C} z$	
PRI13	$x.\ y \lhd_{S>C} z$	$= (x \lhd_{S>C} z).\ (y \lhd_{S>C} \delta)$	
PRI14	$(x + y) \lhd_{S>C} z$	$= (x \lhd_{S>C} y) \lhd_{S>C} z + (y \lhd_{S>C} x) \lhd_{S>C} z$	
PRI15	$x \lhd_{S>C} (\phi \rightarrow y)$	$= \phi \rightarrow (x \lhd_{S>C} y) + \neg\phi \rightarrow (x \lhd_{S>C} \delta)$	
PRI16	$(\phi \rightarrow x) \lhd_{S>C} y$	$= \phi \rightarrow (x \lhd_{S>C} y)$	
PRI17	$block(S, a)$	$= \delta$	if $a \in S$
PRI18	$block(S, a)$	$= a$	otherwise
PRI19	$block(S > C, a)$	$= \delta$	if $a \in S$
PRI20	$block(S > C, a)$	$= block(C, a)$	otherwise
PRI21	$block(C, x + y)$	$= block(C, x) + block(C, y)$	
PRI22	$block(C, x.\ y)$	$= block(C, x).\ y$	

TABLE 2-3. Algebraic laws for prio

Here, *atoms* denotes the set of all atoms, S a set of atoms, and C a chain, i.e. 1 or more sets separated by >.

These laws are derived from [BaeBerKlo86], taking into account the particular definition of a partial order on actions presented here.

In the definition of the priority operator, an auxiliary operator $\lhd_{S>C}$ is used. This operator is parameterized with the priorities. It serves to calculate the context of the first action of its left-hand side. This context, that is the collection of all alternative actions, is being built up in the right-hand side. If some action has low priority and its context can do an action with high priority, this low priority action is blocked. The auxiliary operator *block* is used to check if an action has lower priority than an other one, in which case it must be blocked.

In table 2-4 the transition rules are given. Here, again the operator *block* is used together with an auxiliary operator $\overline{prio}$. It differs from *prio* in that $\overline{prio}(S, x)$ means x, and $prio(S, x)$ means that atoms in set S have higher priority than atoms not in this set. In *pri*9 and *pri*10 the

pri1	$\dfrac{a \notin S}{block(S,a) \xrightarrow{a} \surd}$	pri2	$\dfrac{block(C,a) \xrightarrow{a} \surd \quad a \notin S}{block(S > C,a) \xrightarrow{a} \surd}$
pri3	$\dfrac{x \xrightarrow{a} y}{\overline{prio}(S,x) \xrightarrow{a} y}$	pri4	$\dfrac{x \xrightarrow{a} \surd}{\overline{prio}(S,x) \xrightarrow{a} \surd}$
pri5	$\dfrac{x \xrightarrow{a} y \quad a \in S}{\overline{prio}(S > C,x) \xrightarrow{a} y}$	pri6	$\dfrac{x \xrightarrow{a} \surd \quad a \in S}{\overline{prio}(S > C,x) \xrightarrow{a} \surd}$
pri7	$\dfrac{x \xrightarrow{a} y \quad block(S > C,a) \xrightarrow{a} \surd}{\overline{prio}(S > C,x) \xrightarrow{a} y}$	pri8	$\dfrac{x \xrightarrow{a} \surd \quad block(S > C,a) \xrightarrow{a} \surd}{\overline{prio}(S > C,x) \xrightarrow{a} \surd}$
pri9	$\dfrac{\overline{prio}(C,x) \xrightarrow{a} y \quad \forall b \in S: x \overset{b}{\not\rightarrow}}{\overline{prio}(S > C,x) \xrightarrow{a} y}$	pri10	$\dfrac{\overline{prio}(C,x) \xrightarrow{a} \surd \quad \forall b \in S: x \overset{b}{\not\rightarrow}}{\overline{prio}(S > C,x) \xrightarrow{a} \surd}$
pri11	$\dfrac{\overline{prio}(S > C,x) \xrightarrow{a} y}{prio(S > C,x) \xrightarrow{a} y}$	pri12	$\dfrac{\overline{prio}(S > C,x) \xrightarrow{a} \surd}{prio(S > C,x) \xrightarrow{a} \surd}$
pri13	$\dfrac{\overline{prio}(S > atoms,x) \xrightarrow{a} y}{prio(S,x) \xrightarrow{a} y}$	pri14	$\dfrac{\overline{prio}(S > atoms,x) \xrightarrow{a} \surd}{prio(S,x) \xrightarrow{a} \surd}$

TABLE 2-4. Transition rules for prio

$\overset{b}{\not\rightarrow}$

denotes that there is no transition in which the action *b* can be executed.

2.2.1 The keyword atoms

To provide in easier specification of priorities, the use of the keyword **atoms** is extended. So far, it could only be used as the basic type of a set. Now, it may be used to denote all atoms in the specification, in any place where a set of atoms may be used. So,

```
sets
    of atoms
        H = atoms \ { a, b }
```

and

```
prio(P > atoms, p)
```

are legal constructions.

3. Adjusting the PSF-Toolkit

Although it may not be of interest how software has to be adapted to changes in PSF, it is good to know how much effort it takes to add new operators to PSF. So we give a description of implementing the interrupt, disrupt, and priority operators, without going into too much details.

3.1 The compiler

The parser of the PSF-compiler has to recognize the new features. This can be done in the same manner as it recognizes other operators, without any difficulties. And since binary process operators are all put in the same form, the rest of the compiler can simply deal with the interrupt and disrupt operators as any other binary process operator. The priority operator looks a lot like the hide and encaps operators, with possibly a few more sets, and so it can be dealt with by the compiler in rather the same manner. All it takes is to make a few routines aware of the fact that there are three more operators.

3.2 TIL

Of course, the Tool Interface Language has to be extended with these three operators. We have special routines for reading and writing of TIL-code, which all programs that deal with TIL, use. So the programs which use these routines at least need recompilation.

3.3 The simulator

Adjusting the simulator is not that easy, because it has to deal with the semantics of the new operators. And there were not any similar constructs. How these are implemented is roughly described in the following two sections.

3.3.1 Interrupts and Disrupts

The interrupt and disrupt operators behave somewhat like the + operator, with some restrictions and additions.

The + operator is simulated by making a process-tree for both operands. All of the processes in these trees are marked. If an atom from one of these trees is executed, the other tree has to be removed. We do this by checking the process for a mark. If it has a mark, we go up in the process-tree looking for the process with a + operator. On finding this, we remove the other branch of the tree and take off the markings of the processes in the remaining branch. We repeat these actions until we get to a process without the marking.

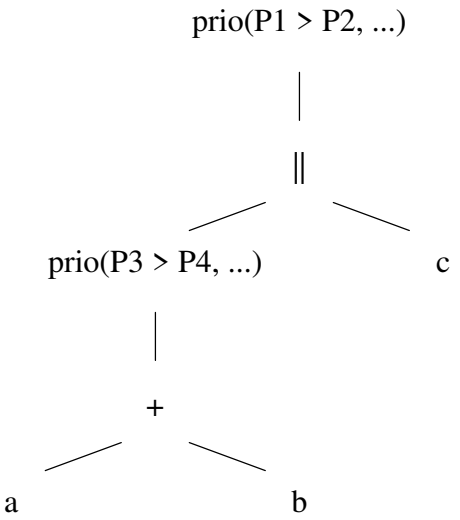
The same can be done for the second argument of the disrupt operator. But if an atom from the process-tree of the first argument is executed, the other process-tree may not be removed. On completion of the execution of the first operand, the tree for the second has to be removed.

If an atom from the process-tree of the second operand of the interrupt operator is executed, there may be no executing of atoms from the first operand until the second is finished. So, we have to disable the process-tree for the first operand. This is done by marking the processes in this tree. Upon completion of the second operand, these markings are removed and a new process-tree for the second operand is made. And as with the disrupt operator, after finishing the execution of the first operand, the tree for the second has to be removed.

3.3.2 Priorities

After we have made the complete process-tree, we calculate the priorities of the atoms roughly as below.

We have a process-tree, for example



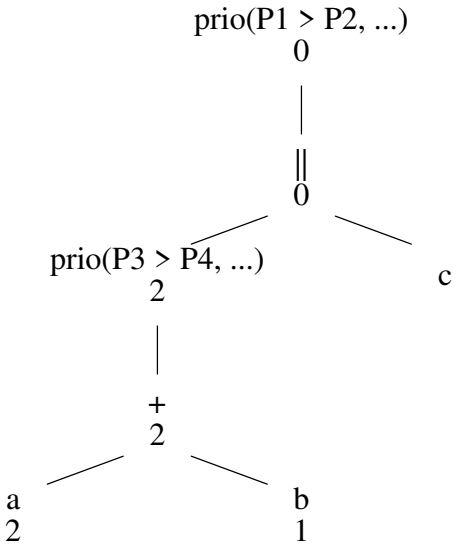
representing the expression $prio(P1 > P2, prio(P3 > P4, a + b) \parallel c)$. Every process has a field added called priority.

First we descend the tree and set all priority-fields to a special value (we use 0 for this), which means no priority. If, on doing this, we find a prio operator when ascending the tree, we calculate the priorities for the atoms in the sub-tree.

We number the sets from right to left beginning with 1, and descend the tree again. Now we check if an atom is a member of one of the sets. We do this from left to right, and assign the number of the first set the atom is a member of, to the priority-field

of the atom. If an atom is not a member of all of these sets we give it the special value to distinguish it from atoms with priority. In doing this, we assign to each process the highest priority of its children.

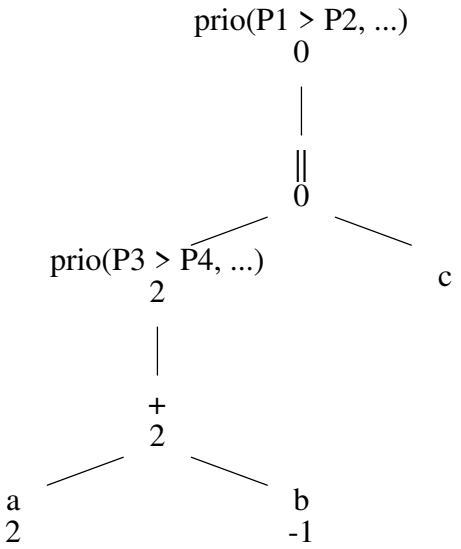
If *a* is member of *P*3 and *b* is member of *P*4 but not of *P*3, this results in the following.



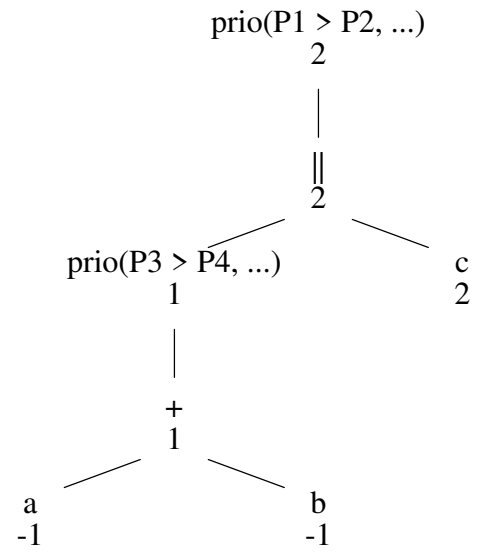
So far, the priority-field of *c* has not got any value assigned, because we deal first with the left operand of ||. first.

Now, we descend the sub-tree again and take along the highest priority. When finding an atom with a priority lower than the highest, we give this atom a negative priority, meaning that it may not be executed.

This results in



After this, we ascend the tree to find the next prio operator (the top one). And we repeat this process. If *c* is a member of *P*1 and *a* of *P*2 but not *P*1, it results in



The result is that we have a process-tree of which the atoms with negative priority may not be executed.

3.4 The verificator

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

4. An example

As an example to show the use of the interrupt, disrupt, and priority operator, we specify a simple machine. In order to concentrate on the use of the three new operators, we don't use any data and put it all into one process module.

The machine is capable of performing some actions in a sequence, that when finished with the last action, are repeated. This machine can be stopped and reset. To control this machine, we have a control-unit with a start-, stop-, continue-, and reset-button.

It is clear that a stop - continue sequence can be modeled by an interrupt, and a reset by a disrupt. So the machine itself can be represented by the process-definition

Machine = start-machine . disrupt(interrupt(Action, Stop), Reset)

In which *Action* is a process that represents the action the machine performs, *Stop* a process that stops the machine and waits for a continue, and *Reset* a process that resets the machine.

To keep things simple we specify

Action = a . b . c . Action

in which *a*, *b*, and *c* are atoms. The stop - continue sequence as

Stop = stop-machine . continue-machine

and

Reset = reset-machine

The control-unit gives us four buttons to control the machine and can be specified by

Control = press-start . do-start . Control-running
Control-running = press-stop . do-stop . Pause + press-reset . do-reset
Pause = press-continue . do-continue . Control-running + press-reset . do-reset

We tie the control-unit and the machine together with the use of communications, and force these communications by means of an encapsulation. Now we have to force a stop or a reset whenever one of these buttons is pressed. We do this by using the priority operator, with which we give a reset and a stop higher priority then the other atoms. This results in the following specification.

```
process module Machine
begin
    atoms
        start-machine, stop-machine, continue-machine, reset-machine
        press-start, press-stop, press-continue, press-reset
        do-start, do-stop, do-continue, do-reset
        start, stop, continue, reset
        a, b, c
    processes
        Start
        Machine
        Action
        Stop
        Reset
        Control
        Control-running
        Pause
    sets
        of atoms
            H = { do-start, do-stop, do-continue, do-reset, start-machine,
                stop-machine, continue-machine, reset-machine }
            P = { reset, stop }
    communications
        do-start | start-machine = start
        do-stop | stop-machine = stop
        do-continue | continue-machine = continue
        do-reset | reset-machine = reset
    definitions
        Start = prio(P > atoms, encaps(H, Machine || Control)) . Start
        Machine = start-machine . disrupt(interrupt(Action , Stop) , Reset)
        Action = a . b . c . Action
        Stop = stop-machine . continue-machine
        Reset = reset-machine
        Control = press-start . do-start . Control-running
        Control-running = press-stop . do-stop . Pause + press-reset . do-reset
        Pause = press-continue . do-continue . Control-running + press-reset . do-reset
end Machine
```

5. References

[BaeBerKlo86] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop, "Syntax and defining equations for an interrupt mechanism in process algebra," in *Fundamenta Informaticae IX (2)*, pp. 127-168, 1986.

[Ber89] J.A. Bergstra, "A mode transfer operator in process algebra," report P8808b, Programming Research Group - University of Amsterdam, 1989.

[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.

[ISO89] International Organization for Standardization, *Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour,* ISO, 1989.

[Mau91] S. Mauw, "PSF - A Process Specification Formalism," Ph.D. thesis, University of Amsterdam, 1991.

[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.