1. Introduction

While we were implementing a simulator for PSF, which is part of the project that aims at the development of a toolkit for the PSF-language, the following question came up:
"Is it possible to design a specification of a simulator for PSF in PSF itself?"

As we shall see in chapter 2, this is possible. In chapter 3 it is shown that the resulting specification can be extended with new operators, that are not yet implemented in PSF as it is. So it gives us a platform to test our ideas on extenions of PSF. The complete specification can be found in appendix A.

The remainder of this chapter gives a short description of PSF and the toolkit developed up till now. This toolkit is heavily used for testing of the simulator.

We like to thank Sjouke Mauw, Jos van Wamel, and Joris Hillebrand for proofreading and their remarks, and the latter also for his design of the cover.

1.1 PSF

PSF (Process Specification Formalism) is a Formal Description Technique developed for the specification of concurrent systems. The formal definition of PSF can be found in [MauVel90]. In [MauVel89a] an introduction to the basic features is given.

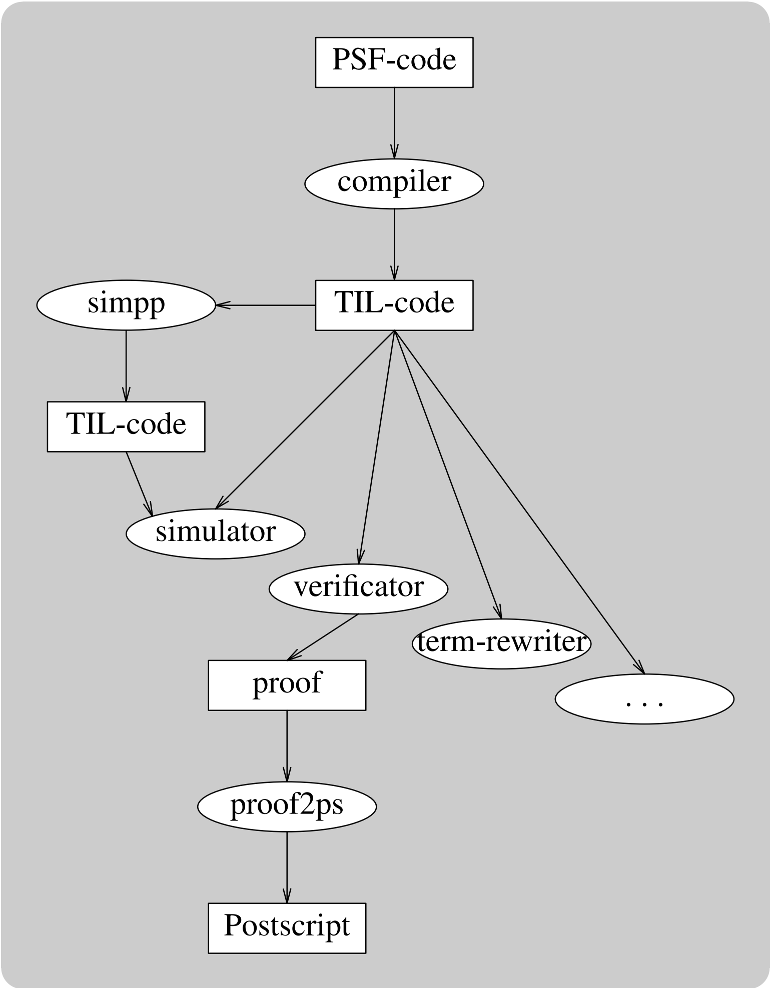
PSF has been designed as the base for a set of tools to support ACP (Algebra of Communicating Processes) [BerKlo86]. It is very close to the informal syntax normally used in denoting ACP-expressions. The part of PSF that deals with the description of data is based on ASF (Algebraic Specification Formalism) [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 only 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. *Specification of a simulator*

First of all, since we want to simulate a specification, the specification has to be expressed in the data-part of PSF. So let's see how we can express a process definition

```
X = a . b
```

with atoms a and b. We can do this by specifying X, a, b, and . as functions, like this

```
sorts
        Process
functions
        _;_     : Process # Process     -> Process
        X       :       -> Process
        a, b    :       -> Process
```

We can't specify a . operator in PSF, because it is not an operator symbol, so we use ; instead. And add the equation

```
X = a ; b
```

Now every occurrence of X will be rewritten into a ; b. But if we have the process definition

```
X = a . X
```

expressed in data in this way, an occurrence of X will result in an endless rewriting of X. So we have to invent a mechanism that prevents this. We can add a function

```
process-def     : Process-> Process
```

and change the equation to

```
process-def(X)    = a ; X
```

If we want to rewrite X we can replace X by *process-def*(X), and we get the process expression a ; X.

But how do we know whether X is an atom or a process. Since atoms and processes are both functions, we don't know this. We can solve this by adding another function

```
process-id      : Process-> Process
```

and changing the equation into

```
process-def(X)  = a ; process-id(X)
```

Now, if we see a *process-id*(X) and we want this replaced by the definition, we simply replace it by *process-def*(X).

Furthermore, we need the possibility to execute an atom or a process, in the process part of the simulator. We can do this as follows:

```
atoms
        A       : Process
processes
        P       : Process
```

If we want to execute atom a, we do this by using $A(a)$, and a process x by using $P(x)$. We can use the following definitions

```
P(process-id(x))  = P(process-def(x))
P(x)              = A(x)
```

with variable $x\!: \rightarrow Process$. But the second definition will also match with *process-id*(x). Somehow we have to decide whether we have to do with an atom. We can do this by adding a function and some equations.

```
functions
        expand  : Process-> Process
        Delta   :         -> Process
equations
[e1]    expand(process-id(x))     = expand(process-def(x))
[e2]    expand(process-def(x))    = Delta
```

and changing the definitions to

```
P(process-id(x))  = P(expand(process-id(x)))
P(expand(x))      = A(x)
```

Note that we rely on the term-rewriting technique. We expect an innermost strategy, so that *process-def*(x) is rewritten before *expand* is taken into account.

The *Delta* is added to manipulate with a deadlock. In PSF, a process identifier for which there is no matching definition, denotes a deadlock. A later version of PSF will be provided with a *Delta*.

The following example gives an idea of how it works. We have the definition

```
process-def(X)  = a ; process-id(X)
```

We simulate process X by

```
P(process-id(X))
```

This gives

```
P(expand(process-id(X)))
```

The term *expand*(*process-id*(X)) is rewritten to *expand*(*process-def*(X)). In which *process-def*(X) is rewritten to a ; *process-id*(X), so that we end up with the term

```
P(expand(a ; process-id(X)))
```

With the additions to the specification given in section 2.2, the term *expand*(a ; *process-id*(X)) is rewritten to a ; *process-id*(X). Leaving us with the process expression

```
P(a ; process-id(X))
```

What matches the definition given in section 2.2, and thus results in

    P(expand(a)) . P(expand(process-id(X)))

The left operand gives us

    A(x)

The following section gives a description of the base of the simulator. The other sections describe how the operators in PSF are to be specified in the data-part of PSF and how they can be simulated.

Note: We expect the specification in the data-part to be a correct specification. There will be no effort made to deal with incorrect specifications.

2.1 The base of the Simulator

We specify the base of the simulator.

```
data module Base
begin
        exports
        begin
                sorts
                        Process
                functions
                        Delta           :           -> Process
                        process-def     : Process-> Process
                        process-id      : Process-> Process
        end
        variables
                x       :       -> Process
        equations
        [r1]    process-def(process-def(x))     = process-def(x)
        [r2]    process-id(process-id(x))       = process-id(x)
        [r3]    process-def(process-id(x))      = process-def(x)
        [r4]    process-id(process-def(x))      = process-id(x)
end Base

data module Operators
begin
        exports
        begin
                functions
                        -- The operators have to be added here.
        end
        imports
                Base
end Operators
```

```
data module Expand
begin
    exports
    begin
        functions
            expand  : Process-> Process
    end
    imports
        Operators
    variables
        x       :       -> Process
    equations
    [e1]    expand(process-id(x))     = expand(process-def(x))
    [e2]    expand(process-def(x))    = Delta
    [e11]   expand(Delta)             = Delta
end Expand

process module Simulator
begin
    exports
    begin
        processes
            Start   : Process
    end
    imports
        Expand
    processes
        P       : Process
    variables
        x       :       -> Process
    definitions
        Start(x)        = P(expand(process-id(x)))
        P(expand(x))    = A(x)
end Simulator
```

The equations in module Base are added to do something useful with errorness specifications in the data-part, but they also limit the number of elements of the sort Process.

The simulator can be used as follows. One makes a specification in data and exports the processes that can be used as start-processes. To do this, a process-module must be added that imports both the specification and the Simulator, and looks like the following.

```
processes
    Start
definitions
    Start = Start(X1) + Start(X2)
```

When $X1$ and $X2$ are the exported start-processes of the specification.

2.2 The . operator

As we have seen above we can specify this operator as follows. We add a function to the module Operators.

```
functions
        _;_      : Process # Process        -> Process
```

To the module Expand we add:

```
equations
[e3]     expand(x ; y)     = x ; y
```

And to the module Simulator:

```
definitions
         P(x ; y)  = P(expand(x)) . P(expand(y))
```

2.3 The + operator

We can specify this operator in a similar way as the . operator. To the module Operators we add the function

```
_+_      : Process # Process        -> Process
```

To the module expand we add the equation

```
[e4]     expand(x + y)     = x + y
```

And to the module Simulator we add the definition

```
P(x + y) = P(expand(x)) + P(expand(y))
```

2.4 The skip operator

We see the skip not as an operator, but more like an atomic action. So we rather add the skip to the module Base then to the module Operators.

```
Skip     :          -> Process
```

To the module Expand we add

```
[e10]    expand(Skip)      = Skip
```

and too the module Simulator

definitions
P(Skip) = **skip**

2.5 The || operator

We could easily do the following.

```
_!!_     : Process # Process        -> Process

[e5]     expand(x !! y)    = x !! y

P(x !! y) = P(expand(x)) || P(expand(y)
```

For some obscure reason PSF does not allow the character | as part of a function name, so we use ! instead. (A later version of PSF, will allow a | as operator symbol.)

But we have to deal here with communication, *encaps*, and *hide* as well. We could specify communications as follows.

```
communications
        A(a) | A(b)        = A(c)
```

This means that this part of the specification to be simulated has to be specified in the process part and not in the data part of PSF. The *encaps* and *hide* operators give more problems. We can't refer to a set in the data part, so we can't make use of the *encaps* operator of PSF for simulating, and thus we have to deal with sets in the data-part of PSF (this will be discussed later). The consequence of this is that we can't make use of the communications in PSF.

In order to solve this, we use the following axiom.

$x \parallel y \quad = x ⫡ y + y ⫡ x + x \mid y$

This implies that we have to do some rewriting in order to get terms without ⫡ and |.

To the module Operators we add the functions

```
_!!_    : Process # Process       -> Process
_!-_    : Process # Process       -> Process       -- leftmerge
_!_     : Process # Process       -> Process       -- communication
```

And to the module Expand

```
functions
        _&!!_   : Process # Process       -> Process
equations
[e5]    expand(x !! y)     = expand(x &!! y)
```

Since we don't want every occurrence of !! to be rewritten, we have to use an extra operator (&!!), to which we can translate !! when we want to expand it.

Now, we have to introduce some equations, which describe how to rewrite the &!!. So we add a new module Rewrite.

```
data module Rewrite
begin
        imports
                Expand
        variables
                x, y, z    :       -> Processes
equations
        [r-mrg]  x &!! y           = (x !- y) + (y !- x) + communicate (x ! y)
end Rewrite
```

The !- and *communicate*(... ! ...) operators will be discussed in the following two sections.

Instead of importing the module Expand in the module Simulator, we now import the module Rewrite.

2.5.1 The !- *operator*

We want to rewrite expressions with the !- operator to expressions that do not contain this operator. For example, we want to have

```
x !- y     = x ; y
```

but this is only allowed when x is an atom. Thus we need someway to decide whether a process is atomic or not. In order to achieve this, we introduce the following modules.

```
data module Booleans
begin
        exports
        begin
                sorts
                        Boolean
                functions
                        true    :           -> Boolean
                        false   :           -> Boolean
                        not     : Boolean       -> Boolean
        end
        equations
        [bool1] not(true)       = false
        [bool1] not(false)      = true
end Booleans

data module Atomic
begin
        exports
        begin
                functions
                        is-atom  : Process-> Boolean
        end
        imports
                Booleans, Operators
        functions
                atomic   : Process-> Boolean
        variables
                x, y     :          -> Process
        equations
        [atomic1]       atomic(x !! y)    = false
        [atomic2]       atomic(x + y)     = false
        [atomic3]       atomic(x ; y)     = false
        [atomic4]       atomic(Skip)      = false
        [atomic5]       atomic(Delta)     = false
        [atomic6]       not(atomic(x))    = false
        [atomic7]       is-atom(x)        = true    when
                                not(atomic(x))    = false
end Atomic
```

Now we can introduce some rewrite rules for the !- operator. We add the following to the module Rewrite.

```
imports
        Atomic
equations
[r-seq1]        (x + y) ; z       = (x ; z) + (y ; z)
[r-seq2]        (x ; y) ; z       = x ; (y ; z)
[r-lmrg1a]      x !- y            = x ; y   when
                        is-atom(x) = true
[r-lmrg1b]      Skip !- y         = Skip ; y
[r-lmrg1c]      Delta !- x        = Delta
[r-lmrg2]       (x ; y) !- z      = x ; (y !! z)      when
                        is-atom(x) = true
[r-lmrg3]       (x + y) !- z      = (x !- z) + (y !- z)
```

The equations [r-seq1] and [r-seq2] introduce a normal form, which makes it unnecessary to have equations for all possible forms.

Sometimes we need to do an expansion, before we can rewrite a term with the !- operator. Therefore we could add the following equations to module Expand.

```
[lmerge1]     process-id(x) !- y          = process-def(x) !- y
[lmerge2]     (x !! y) !- z               = (x &!! y) !- z
[lmerge3]     (process-id(x) ; y) !- z    = (process-def(x) ; y) !- z
```

We need an equation for every possible term that has to be rewritten, when it appears as the left-hand-side of the !- operator. As we shall see later on, we have to do this for other operators as well. It is better to do this as follows. We make a new module.

```
data module Termtype
begin
    exports
    begin
        functions
            rewrite-term    : Process-> Boolean
    end
    imports
        Operators, Booleans
    variables
        x, y    :    -> Process
    equations
    [rew1]  rewrite-term(process-id(x))    = true
    [rew2]  rewrite-term(x !! y)           = true
end
```

And add the following equation to module expand.

```
[lmerge1]     x !- y          = expand(x) !- y   when
                   rewrite-term(x)   = true
[lmerge2]     (x ; y) !- z    = (expand(x) ; y) !- z     when
                   rewrite-term(x)   = true
```

Now that we have this scheme, we can use this in module Atomic as well. We therefore change equation [atomic1] to:

```
[atomic1]     atomic(x)       = false   when
                   rewrite-term(x)   = true
```

And import the module Termtype instead of the modules Booleans and Operators.

But, $expand(x)$ can result in $expand(a)$, (a is an atom), where we only wanted a. We can solve this special case by adding the following. To the module Base

```
functions
    atom    : Process-> Process
```

To the module Termtype

```
equations
[rew0]  rewrite-term(atom(x))     = true
```

To the module Expand

```
equations
[es1]   expand(x)         = atom(x)        when
            is-atom(x)         = true
[es2]   expand(atom(x))   = x
```

And change the definition

```
P(expand(x))      = A(x)
```

in the module Simulator in

```
P(atom(x))          = A(x)
```

Now, when atom *a* is the result of an expansion, it appears as *atom*(*a*). When it is a subterm, it is rewritten to *a*.

2.5.2 *The* communicate(... ! ...) *operator*

We expect communications to be written in data like

```
a ! b     = c
```

which means that *a* and *b* communicate to *c*. If no communication is specified for *a* and *b*, it should be rewritten to a deadlock, in order to make other rewritings possible. So we need an extra operator (communicate). We add the following to the module Expand:

```
functions
        communicate     : Process-> Process
```

And module Rewrite

```
equations
[r-comm1]    communicate(x ! y)      = Delta
[r-comm2]    communicate(x)          = x          when
                   is-atom(x)        = true
[r-cmm1a]    (x ; y)   ! z           = communicate(x ! z) ; y   when
                   is-atom(x)        = true,
                   is-atom(z)        = true
[r-cmm1b]    x ! (y ; z)             = communicate(x ! y) ; z   when
                   is-atom(x)        = true,
                   is-atom(y)        = true
[r-cmm2]     (x ; y) ! (z ; w)       = communicate(x ! z) ; (y !! w)      when
                   is-atom(x)        = true,
                   is-atom(x)        = true
[r-cmm3a]    (x + y) ! z             = communicate(x ! z) + communicate(y ! z)
[r-cmm3b]    x ! (y + z)             = communicate(x ! y) + communicate(x ! z)
[r-cmm4a]    Delta ! x               = Delta
[r-cmm4b]    x ! Delta               = Delta
```

And some equations to module Expand.

```
[comm1]      x ! y              = expand(x) ! y   when
                   rewrite-term(x)   = true
[comm2]      x ! y              = x ! expand(y)   when
                   rewrite-term(y)   = true
[comm3]      (x ; y) ! z        = (expand(x) ; y) ! z        when
                   rewrite-term(x)   = true
[comm4]      x ! (y ; z)        = x ! (expand(y) ; z)        when
                   rewrite-term(y)   = true
```

2.6 *Sets*

As mentioned earlier, we have no way of referencing to a set in the data-part of PSF. So we have to specify a construction for this. First, we specify a construction for enumerated sets. We add the sort Set to module Base and make a new module.

```
data module Sets
begin
        exports
        begin
                functions
                        cons        : Set # Set        -> Set
                        NIL         :                  -> Set
                        element-of  : Set # Process    -> Boolean
                        el          : Process-> Set
        end
        imports
                Base, Booleans
        variables
                x           :           -> Process
                l1, l2      :           -> Set
        equations
        [set1]  element-of(cons(el(x), l1), x)      = true
        [set2]  element-of(cons(l1, l2), x)         = true   when
                        element-of(l2, x)  = true
        [set3]  not(element-of(l1, x))              = true
end Sets
```

The function *el* is used as a typecast, so if we want to make use of *cons* for an other type (than atoms), we only have to introduce a new function *el*.

We can now define a set H consisting of the atoms a, b, and c as an equation, like this:

```
functions
        a, b, c    :        -> Process
        H          :        -> Set
equations
[example]          H        = cons(el(a), cons(el(b), cons(el(c), NIL)))
```

If we want to know if the condition *element-of*(H, b) holds, the following rewritings are done.

```
element-of(H, b)
element-of(cons(el(a), cons(el(b), cons(el(c), NIL))), b)               [example]
element-of(cons(el(a), cons(el(b), cons(el(c), NIL))), b) = true when
        element-of(cons(el(b), cons(el(c), NIL)), b) = true             [set2]
        true = true                                                     [set1]
true
```

The term *not*(*element-of*(H, c)) is rewritten to *false*, and *not*(*element-of*(H, d)) to *true*. So we now have the possibility to decide whether an atom is part of a set or not.

We now specify the other set constructors (union, intersection, difference), by adding the following to module Sets.

```
functions
            _+_      : Set # Set       -> Set
            _;_      : Set # Set       -> Set
            _\_      : Set # Set       -> Set
equations
[set4]    element-of(l1 + l2, x)      = true    when
                element-of(l1, x)  = true
[set5]    element-of(l1 + l2, x)      = true    when
                element-of(l2, x)  = true
[set6]    element-of(l1 ; l2)         = true    when
                element-of(l1, x)  = true,
                element-of(l2, x)  = true
[set7]    element-of(l1 \ l2)         = true    when
                element-of(l1, x)  = true,
                not(element-of(l2, x))    = true
```

Only the placeholder construction, as in

$$H = \{ f(t) \mid t \text{ in } S \}$$

has not been specified yet. This gives a problem, because it introduces a new variable, which is something we cannot do in a equation. The only way to see whether x is an element of the set H is by matching x with $f(t)$. Consider the following equation.

```
[set8]    element-of(placeholder(l1), x)        = true    when
                placeholder-match(l1, x)   = true
```

We can now specify a placeholder by doing the following.

```
functions
            PH1      :          -> Set
equations
[PH1]     placeholder-match(PH1, f(t))           = true
```

And using *placeholder*$(PH1)$ in the construction of a set.

So we add the following to the module Sets.

```
functions
            placeholder        : Set     -> Set
            placeholder-match            : Set # Process    -> Boolean
equations
[set8]    element-of(placeholder(l1), x)        = true    when
                placeholder-match(l1, x)   = true
```

2.7 The encaps operator

Now that we have a construction for sets in the data part of PSF, we can specify the *encaps* operator. To the module Operators we add

```
functions
            Encaps   : Set # Process     -> Process
```

To the module Termtype

```
variables
            l        :          -> Set
equations
            [rew3]   rewrite-term(Encaps(l, x)) = true
```

And to the module Expand

```
functions
            iEncaps  : Set # Process     -> Process
variables
            l            :             -> Set
equations
[e6]        expand(Encaps(l, x))        = iEncaps(l, x)
[enc]       iEncaps(l, x)               = iEncaps(l, expand(x))     when
                        rewrite-term(x)     = true
```

And to the module Rewrite

```
variables
            l1           :             -> Set
equations
[r-enc1]  iEncaps(l1, x)       = Delta   when
                        is-atom(x)            = true,
                        element-of(l1, x)  = true
[r-enc2]  iEncaps(l1, x)       = x         when
                        is-atom(x)            = true,
                        not(element-of(l1, x))        = true
[r-enc3]  iEncaps(l1, x + y)= Encaps(l1, x) + Encaps(l1, y)
[r-enc4]  iEncaps(l1, x ; y) = Encaps(l1, x) ; Encaps(l1, y)
```

2.8 *The hide operator*

We specify the *hide* operator in the same way as the *encaps* operator. To the module Operators we add

```
functions
            Hide       : Set # Process     -> Process
```

To the module Termtype

```
[rew4]    rewrite-term(Hide(l, x))      = true
```

And to the module Expand

```
functions
            iHide      : Set # Process     -> Process
variables
            l            :             -> Set
equations
[e7]        expand(Hide(l, x))          = iHide(l, x)
[hid]       iHide(l, x)                 = iHide(l, expand(x))        when
                        rewrite-term(x)     = true
```

And to the module Rewrite

```
variables
        l1          :           -> Set
equations
[r-hid1]  iHide(l1, x)        = Skip   when
                is-atom(x)           = true,
                element-of(l1, x)    = true
[r-hid2]  iHide(l1, x)        = x      when
                is-atom(x)           = true,
                not(element-of(l1, x))      = true
[r-hid3]  iHide(l1, x + y)    = Hide(l1, x) + Hide(l1, y)
[r-hid4]  iHide(l1, x ; y)    = Hide(l1, x) ; Hide(l1, y)
```

2.9 The sum operator

Consider the following.

$$sum(v\ in\ S, P(v))$$

If S denotes an enumeration $\{v_1, v_2, \ldots, v_n\}$, then this is an abbreviation of

$$P(v_1) + P(v_2) + \cdots + P(v_n)$$

But if S is not a finite sort or set, then we have something that we cannot rewrite to a finite term. So this construction may be nice in theory, but in practice it is useless, if S is not finite.

If S is finite, S can be replaced by a set that solely consist of an enumeration. So we have decided to specify the sum operator only for an enumeration.

Let's see how we have to specify a sum construction in data. We add to the module Operators

```
functions
        Sum     : Set # Process     -> Process
        In      : Set # Set         -> Set
```

We want to specify the definition

X = **sum**(x **in** H, Y(x))

in data.

```
sorts
        D
functions
        Y        : D        -> Process
        X        :          -> Process
        a, b, c  :          -> D
        el       : D        -> Set
variables
        x        :          -> D
equations
[set]   H                   = cons(el(a), cons(el(b), cons(el(c), NIL)))
[def]   process-def(X)      = Sum(In(el(x), H), Y(x))
```

This introduces a variable on the right side of equation [def], which is not allowed. Somehow, we have to get rid of that free variable. We try the following. We add to the module Operators

```
functions
        Sum      : Set # Process    -> Process
```

to the module Termtype

```
[rew5]  rewrite-term(Sum(l, x))     = true
```

and to the module Expand

```
functions
        iSum     : Set # Process    -> Process
equations
[e8]    expand(Sum(l, x))           = expand(iSum(l, x))
```

and to the module Sets

```
functions
        fill-in  : Set # Process    -> Process
```

and to the module Rewrite

```
equations
[r-sum1] iSum(cons(l1, NIL), x)              = fill-in(l1, x)
[r-sum2] iSum(cons(l1, cons(l2, l3)), x)     = fill-in(l1, x) + Sum(cons(l2, l3), x)
[r-sum3] iSum(NIL, x)                        = Delta
```

and specify the definition as follows

```
sorts
        D
functions
        Y        : D      -> Process
        X        :        -> Process
        a, b, c  :        -> D
        el       : D      -> Set
variables
        x        :        -> D
equations
[set]   H                           = cons(el(a), cons(el(b), cons(el(c), NIL)))
[SE1]   fill-in(el(x), SUMEXPR1)    = Y(x)
[def]   process-def(X)              = Sum(H, SUMEXPR1)
```

2.10 The merge operator

The *merge* can be specified in the same way as the *sum*. We add to the module Operators

```
functions
        Merge    : Set # Process    -> Process
```

to the module Termtype

```
[rew6]  rewrite-term (Merge(l, x))  = true
```

and to the module Expand

```
functions
        iMerge   : Set # Process    -> Process
equations
[e9]    expand(Merge(l, x))         = expand(iMerge(l, x))
```

and to the module Rewrite

equations

[r-mrg1] iMerge(cons(l1, NIL), x)	= fill-in(l1, x)
[r-mrg2] iMerge(cons(l1, cons(l2, l3)), x)	= fill-in(l1, x) !! Merge(cons(l2, l3), x)
[r-mrg3] iMerge(NIL, x)	= Delta

2.11 Extra rewrite rules

To the module Rewrite we add

[r-delta1]	x + Delta	= x
[r-delta2]	Delta + x	= x
[r-delta3]	Delta ; x	= Delta
[r-skip1]	x ; Skip	= x
[r-skip2]	(Skip ; x) + x	= Skip ; x
[r-skip1']	x ; (Skip ; y)	= x ; y

The above rewrite rules are not really necessary, but they make the terms smaller and so speed up the rewrite system.

3. *Simulation of extensions of PSF*

Now that we have specified a simulator for PSF, we can use it as a platform for simulating extensions of PSF. In the following sections we shall specify a few possible extensions.

3.1 Conditional Choices

We want to extend PSF with a mechanism to control the flow of a process. For example:

```
P(x, y) = a(x) . if x = y then b(x) else c(x)
```

First we make an *if* construction. We introduce a function *If* with three arguments and add the following equation.

```
If(x, y, P)         = P       when
          x         = y
```

or

```
If(x, x, P)         = P
```

This part was easy, but when the condition fails, the result has to be a deadlock. There is no possibility to test for inequality in PSF, so we have to invent something for this. We add the function *equal* with the following equations.

```
equal(x, x)         = true
not(equal(x, y))    = true
```

And now we can add the equation:

```
If(x, y, P)         = Delta   when
          not(equal(x, y))    = true
```

In the same way, we can specify an *if-else* construction. We introduce for this a function *If-else* with four arguments and the equations:

```
If-else(x, x, PA, PB)       = PA
If-else(x, y, PA, PB)       = PB      when
        not(equal(x, y))    = true
```

We can also express the *if* in *if-else*, like this:

```
If(x, y, P)          = If-else(x, y, P, Delta)
```

What reduces the number of equations we have to add for rewriting.

To specify the *if* and *if-else* we add the following.

```
data module If
begin
        exports
        begin
                sorts
                        Data
                functions
                        If        : Data # Data # Process      -> Process
                        If-else   : Data # Data # Process # Process    -> Process
                        equal     : Data # Data        -> Boolean
        end
        imports
                Base, Booleans
        variables
                x, y       :          -> Data
        equations
        [eq1]   equal(x, x)        = true
        [eq2]   not(equal(x, y))   = false
end If
```

In the module Termtype we import the module If and add the following.

```
variables
        d1, d2    :          -> Data
equations
[rew7]  rewrite-term(If(d1, d2, x)) = true
[rew8]  rewrite-term(If(d1, d2, x, y))         = true
```

To the module Expand we add

```
exports
begin
        functions
                iIf-else   : Data # Data # Process      -> Process
end
variables
        d1, d2    :          -> Data
equations
[e12]   expand(If(d1, d2, x))                 = expand(iIf-else(d1, d2, x, Delta))
[e13]   expand(If-else(d1, d2, x, y))         = expand(iIf-else(d1, d2, x, y))
```

And to the module Rewrite

```
equations
[r-if1] iIf-else(d1, d1, x, y)      = expand(x)
[r-if2] iIf-else(d1, d2, x, y)      = expand(y)        when
                not(equal(x, y))    = true
```

We have specified the *if* and *if-else* constructions for the sort *Data*. For other types, a typecast has to be used like we did with the sets.

3.2 Priorities

In [Mau91] priorities were suggested as an extension of PSF. Therefore, a new operator was introduced with two arguments, the first being a set of atomic actions and the second a process-expression, like this:

$$prio(\{a, b\}, a + b.c)$$

The atomic actions that are member of the set have priority over other actions. Thus it means that if an expression that has an alternative that starts with an atom with high priority, alternatives that start with an atom with low priority are suppressed. So for $c \neq a$ and $c \neq b$, the expression $prio(\{a\}, a + b)$ equals a, and $prio(\{c\}, a + b)$ equals $a + b$.

For manipulating with this new operator, we introduce a second one *iprio*, with three arguments, first the set of atomic actions, and the other two process-expressions. The semantics of these operators are given by the following equations, in which a and b are atomic actions and *a* may not be equal to *skip*, *x*, *y*, and *z* are process-expressions, and *S* is a set of atomic actions of which *skip* is not an element.

PRI1	prio(S, x)	= iprio(S, x, δ)	
PRI2	iprio(S, a, b)	= a	if $a \in S \vee b \notin S$
PRI3	iprio(S, a, b)	= δ	otherwise
PRI4	iprio(S, a, δ)	= a	
PRI5	iprio(S, δ, a)	= δ	
PRI6	iprio(S, a, skip)	= a	
PRI7	iprio(S, skip, a)	= skip	
PRI8	iprio(S, x, y . z)	= iprio(S, x, y)	
PRI9	iprio(S, x, y + z)	= iprio(S, iprio(S, x, y), z)	
PRI10	iprio(S, x . y, z)	= iprio(S, x, z) . iprio(S, y, δ)	
PRI11	iprio(S, x + y, z)	= iprio(S, iprio(S, x, y), z) + iprio(S, iprio(S, y, x), z)	

This can easily be specified as follows.

```
data module Priorities
begin
        exports
        begin
                functions
                        Prio        : Set # Process     -> Process
        end
        imports
                Base
end Priorities
```

In the module Termtype we import the module Priorities and add the following.

```
variables
        l       :       -> Set
equations
[rew9]  rewrite-term(Prio(l, x))     = true
```

To the module Expand we add

```
exports
begin
        iPrio      : Set # Process # Process   -> Process
end
equations
[e14]   expand(Prio(l, x))= expand(iPrio(l, x, Delta))
[prio1] iPrio(l, x, y)      = iPrio(l, expand(x), y)     when
                rewrite-term(x)   = true
[prio2] iPrio(l, x, y)      = iPrio(l, x, expand(y))     when
                rewrite-term(x)   = true
```

And to the module Rewrite

```
equations
[r-prio1]     iPrio(l1, x, y)      = x       when
                   is-atom(x)          = true,
                   is-atom(y)          = true,
                   element-of(l1, x)   = true
[r-prio2]     iPrio(l1, x, y)      = x       when
                   is-atom(x)          = true,
                   is-atom(y)          = true,
                   not(element-of(l1, y))      = true
[r-prio3]     iPrio(l1, x, y)      = Delta   when
                   is-atom(x)          = true,
                   is-atom(y)          = true,
                   not(element-of(l1, x))      = true,
                   element-of(l1, y)   = true
[r-prio4]     iPrio(l1, x, Delta)= x         when
                   is-atom(x)          = true
[r-prio5]     iPrio(l1, Delta, x)= Delta
[r-prio6]     iPrio(l1, x, Skip)  = x        when
                   is-atom(x)          = true
[r-prio7]     iPrio(l1, Skip, x)  = Skip
[r-prio8]     iPrio(l1, x, y ; z)  = iPrio(l1, x, y)
[r-prio9]     iPrio(l1, x, y + z) = iPrio(l1, iPrio(l1, x, y), z)
[r-prio10]    iPrio(l1, x ; y, z)  = iPrio(l1, x, z) ; iPrio(l1, y, Delta)
[r-prio11]    iPrio(l1, x + y, z)  = iPrio(l1, iPrio(l1, x, y), z) + iPrio(l1, iPrio(l1, y, x), z)
```

4. References

[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.

[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.

[MauVel89a] S. Mauw and G.J. Veltink, "An introduction to PSF," in *Proc. International Joint Conference on Theory and Practice of Software Development, TAPSOFT '89*, ed. J. Diaz, F. Orejas, eds., LNCS 352, pp. 272-285, Springer Verlag, 1989.

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

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

A. The specification

```
data module Base
begin
    exports
    begin
        sorts
            Process,
            Set
        functions
            Skip           :          -> Process
            Delta          :          -> Process
            atom           : Process-> Process
            process-id     : Process-> Process
            process-def    : Process-> Process
    end
    variables
        x       :       -> Process
    equations
    [r1]    process-def(process-def(x))       = process-def(x)
    [r2]    process-id(process-id(x))         = process-id(x)
    [r3]    process-def(process-id(x))        = process-def(x)
    [r4]    process-id(process-def(x))        = process-id(x)
end Base
```

```
data module Booleans
begin
        exports
        begin
                sorts
                        Boolean
                functions
                        true    :               -> Boolean
                        false   :               -> Boolean
                        not     : Boolean       -> Boolean
        end
        equations
        [bool1] not(true)       = false
        [bool2] not(false)      = true
end Booleans

data module Operators
begin
        exports
        begin
                functions
                        _+_     : Process # Process     -> Process
                        _;_     : Process # Process     -> Process
                        _!!_    : Process # Process     -> Process

                        _!-_    : Process # Process     -> Process
                        _!_     : Process # Process     -> Process

                        Encaps  : Set # Process -> Process
                        Hide    : Set # Process -> Process

                        Sum     : Set # Process -> Process
                        Merge   : Set # Process -> Process
        end
        imports
                Base
end Operators

data module If
begin
        exports
        begin
                sorts
                        Data
                functions
                        If      : Data # Data # Process  -> Process
                        If-else : Data # Data # Process # Process -> Process
                        equal   : Data # Data   -> Boolean
        end
        imports
                Base, Booleans
        variables
                x, y    :       -> Data
        equations
        [eq1]   equal(x, x)             = true
        [eq2]   not(equal(x, y))        = true
end If
```

```
data module Priorities
begin
        exports
        begin
                functions
                        Prio        : Set # Process     -> Process
        end
        imports
                Base
end Priorities

data module Termtype
begin
        exports
        begin
                functions
                        rewrite-term        : Process-> Boolean
                        not-rewrite-term    : Process-> Boolean
        end
        imports
                Operators, If, Priorities
        variables
                x, y      :         -> Process
                l         :         -> Set
                d1, d2    :         -> Data
        equations
        [rew0]    rewrite-term(atom(x))                 = true
        [rew1]    rewrite-term(process-id(x))           = true
        [rew1']   rewrite-term(process-def(x))          = true
        [rew2]    rewrite-term(x !! y)                  = true
        [rew3]    rewrite-term(Encaps(l, x))            = true
        [rew4]    rewrite-term(Hide(l, x))              = true
        [rew5]    rewrite-term(Sum(l, x))               = true
        [rew6]    rewrite-term(Merge(l, x))             = true
        [rew7]    rewrite-term(If(d1, d2, x))           = true
        [rew8]    rewrite-term(If-else(d1, d2, x, y))   = true
        [rew9]    rewrite-term(Prio(l, x))              = true
        [nrew1]   not(rewrite-term(x))                  = true
end Termtype
```

```
data module Atomic
begin
        exports
        begin
                functions
                        is-atom  : Process-> Boolean
        end
        imports
                Termtype
        functions
                atomic  : Process-> Boolean
        variables
                x, y    :       -> Process
        equations
        [atomic1]       atomic(x)          = false   when
                                rewrite-term(x)   = true
        [atomic2]       atomic(x + y)      = false
        [atomic3]       atomic(x ; y)      = false
        [atomic4]       atomic(Skip)       = false
        [atomic5]       atomic(Delta)      = false
        [atomic6]       not(atomic(x))     = false
        [atomic7]       is-atom(x)         = true    when
                                not(atomic(x))    = false
end Atomic

data module Expand
begin
        exports
        begin
                functions
                        expand          : Process-> Process
                        _&!!_           : Process # Process          -> Process
                        iEncaps         : Set # Process    -> Process
                        iHide           : Set # Process    -> Process
                        iSum            : Set # Process    -> Process
                        iMerge          : Set # Process    -> Process
                        communicate     : Process-> Process
                        iIf-else        : Data # Data # Process # Process   -> Process
                        iPrio           : Set # Process # Process   -> Process
        end
        imports
                Atomic
        variables
                x, y, z   :       -> Process
                l         :       -> Set
                d1, d2    :       -> Data
        equations
        [es1]           expand(x)                  = atom(x)            when
                                is-atom(x)         = true
        [es2]           expand(atom(x))            = x
        [e1]            expand(process-id(x))      = expand(process-def(x))
        [e2]            expand(process-def(x))     = Delta
        [e3]            expand(x ; y)              = x ; y
        [e4]            expand(x + y)              = x + y
        [e5]            expand(x !! y)             = expand(x &!! y)
        [e6]            expand(Encaps(l, x))       = expand(iEncaps(l, x))
        [e7]            expand(Hide(l, x))         = expand(iHide(l, x))
```

```
[e8]        expand(Sum(l, x))            = expand(iSum(l, x))
[e9]        expand(Merge(l, x))          = expand(iMerge(l, x))
[e10]       expand(Skip)                 = Skip
[e11]       expand(Delta)                = Delta
[e12]       expand(If(d1, d2, x))        = expand(iIf-else(d1, d2, x, Delta))
[e13]       expand(If-else(d1, d2, x, y))      = expand(iIf-else(d1, d2, x, y))
[e14]       expand(Prio(l, x))           = expand(iPrio(l, x, Delta))
[lmerge1]   x !- y                       = expand(x) !- y  when
                  rewrite-term(x) = true
[lmerge2]   (x ; y) !- z                 = (expand(x) ; y) !- z      when
                  rewrite_term(x)   = true
[comm1]     x ! y                        = expand(x) ! y   when
                  rewrite-term(x)   = true
[comm2]     x ! y                        = x ! expand(y)   when
                  rewrite-term(y)   = true
[comm3]     (x ; y) ! z                  = (expand(x) ; y) ! z when
                  rewrite-term(x)   = true
[comm4]     x ! (y ; z)                  = x ! (expand(y) ; z)       when
                  rewrite-term(x) = true
[enc]       iEncaps(l, x)                = iEncaps(l, expand(x))     when
                  rewrite-term(x)   = true
[hid]       iHide(l, x)                  = iHide(l, expand(x))       when
                  rewrite-term(x)   = true
[prio1]     iPrio(l, x, y)               = iPrio(l, expand(x), y)    when
                  rewrite-term(x)   = true
[prio2]     iPrio(l, x, y)               = iPrio(l, x, expand(y))    when
                  rewrite-term(y)   = true
```

end Expand

```
data module Sets
begin
        exports
        begin
                functions
                        element-of              : Set # Process   -> Boolean
                        el                      : Process-> Set
                        cons                    : Set # Set       -> Set
                        NIL                     :       -> Set
                        fill-in                 : Set # Process   -> Process
                        _+_                     : Set # Set       -> Set
                        _;_                     : Set # Set       -> Set
                        _ _                     : Set # Set       -> Set
                        placeholder             : Set   -> Set
                        placeholder-match       : Set # Process   -> Boolean
        end
        imports
                Base, Booleans
        variables
                x, y    :       -> Process
                l1, l2  :       -> Set
        equations
        [set1]  element-of(cons(el(x), l1), x)      = true
        [set2]  element-of(cons(l1, l2), x)         = true   when
                        element-of(l2, x)  = true
        [set3]  not(element-of(l1, x))              = true
        [set4]  element-of(l1 + l2, x)              = true   when
                        element-of(l1, x)  = true
        [set5]  element-of(l1 + l2, x)              = true   when
                        element-of(l2, x)  = true
        [set6]  element-of(l1 ; l2, x)              = true   when
                        element-of(l1, x)  = true,
                        element-of(l2, x)  = true
        [set7]  element-of(l1  l2, x)               = true   when
                        element-of(l1, x)  = true,
                        not(element-of(l2, x))      = true
        [set8]  element-of(placeholder(l1), x)      = true   when
                        placeholder-match(l1, x)    = true
        [set9]  element-of(l1, Skip)                = false
        [set10] element-of(l1, Delta)               = false
end Sets
```

```
data module Rewrite
begin
        imports
                Expand,
                Sets
        variables
                w, x, y, z:         -> Process
                l1, l2, l3 :        -> Set
                d1, d2    :         -> Data
        equations
                -- sequential --
        [r-seq1]        (x + y) ; z         = (x ; z) + (y ; z)
        [r-seq2]        (x ; y) ; z         = x ; (y ; z)
                -- merge --
        [r-mrg]         x &!! y             = (x !- y) + (y !- x) + communicate(x ! y)
                -- leftmerge --
        [r-lmrg1a]      x !- y              = x ; y    when
                                is-atom(x)          = true
        [r-lmrg1b]      Skip !- y           = Skip ; y
        [r-lmrg1c]      Delta !- x          = Delta
        [r-lmrg2]       (x ; y) !- z        = x ; (y !! z)        when
                                is-atom(x)          = true
        [r-lmrg3]       (x + y) !- z        = (x !- z) + (y !- z)
                -- communication --
        [r-cmm1a]       (x ; y) ! z         = communicate(x ! z) ; y   when
                                is-atom(x)          = true,
                                is-atom(z)          = true
        [r-cmm1b]       x ! (y ; z)         = communicate(x ! y) ; z   when
                                is-atom(x)          = true,
                                is-atom(y)          = true
        [r-cmm2]        (x ; y) ! (z ; w)   = communicate(x ! z) ; (y !! w) when
                                is-atom(x)          = true,
                                is-atom(z)          = true
        [r-cmm3a]       (x + y) ! z         = communicate(x ! z) + communicate(y ! z)
        [r-cmm3b]       x ! (y + z)         = communicate(x ! y) + communicate(x ! z)
        [r-cmm4a]       Delta ! x           = Delta
        [r-cmm4b]       x ! Delta           = Delta
                -- encaps --
        [r-enc1]        iEncaps(l1, x)      = Delta  when
                                is-atom(x)          = true,
                                element-of(l1, x)   = true
        [r-enc2]        iEncaps(l1, x)      = x      when
                                is-atom(x)          = true,
                                not(element-of(l1, x))      = true
        [r-enc3]        iEncaps(l1, x + y)= Encaps(l1, x) + Encaps(l1, y)
        [r-enc4]        iEncaps(l1, x ; y) = Encaps(l1, x) ; Encaps(l1, y)
                -- hide --
        [r-hid1]        iHide(l1, x)         = Skip   when
                                is-atom(x)          = true,
                                element-of(l1, x)   = true
        [r-hid2]        iHide(l1, x)        = x      when
                                is-atom(x)          = true,
                                not(element-of(l1, x))      = true
        [r-hid3]        iHide(l1, x + y)    = Hide(l1, x) + Hide(l1, y)
        [r-hid4]        iHide(l1, x ; y)    = Hide(l1, x) ; Hide(l1, y)
                -- deadlock --
```

```
[r-delta1]    x + Delta         = x
[r-delta2]    Delta + x         = x
[r-delta3]    Delta ; x         = Delta
        -- skip --
[r-skip1]     x ; Skip          = x
[r-skip2]     (Skip ; x) + x    = Skip ; x
[r-skip1']    x ; (Skip ; y)    = x ; y
        -- sum --
[r-sum1]      iSum(cons(l1, NIL), x)    = fill-in(l1, x)
[r-sum2]      iSum(cons(l1, cons(l2, l3)), x)    = fill-in (l1, x) +
                  Sum(cons(l2, l3), x)
[r-sum3]      iSum(NIL, x)      = Delta
        -- merge --
[r-mrg1]      iMerge(cons(l1, NIL), x)   = fill-in(l1, x)
[r-mrg2]      iMerge(cons(l1, cons(l2, l3)), x)    = fill-in (l1, x) !! Merge(cons(l2, l3), x)
[r-mrg3]      iMerge(NIL, x)    = Delta
[r-comm1]     communicate(x ! y)        = Delta
[r-comm2]     communicate(x)    = x     when
                  is-atom(x)            = true
[r-comm3]     communicate(Delta)        = Delta
[r-comm4]     communicate(Skip)         = Skip
        -- if --
[r-if1]       iIf-else(d1, d1, x, y)    = expand(x)
[r-if2]       iIf-else(d1, d2, x, y)    = expand(y)      when
                  not(equal(d1, d2))          = true
        -- priorities --
[r-prio1]     iPrio(l1, x, y)   = x     when
                  is-atom(x)            = true,
                  is-atom(y)            = true,
                  element-of(l1, x)     = true
[r-prio2]     iPrio(l1, x, y)   = x     when
                  is-atom(x)            = true,
                  is-atom(y)            = true,
                  not(element-of(l1, y))      = true
[r-prio3]     iPrio(l1, x, y)   = Delta when
                  is-atom(x)            = true,
                  is-atom(y)            = true,
                  not(element-of(l1, x))      = true,
                  element-of(l1, y)     = true
[r-prio4]     iPrio(l1, x, Delta) = x   when
                  is-atom(x)            = true
[r-prio5]     iPrio(l1, Delta, x) = Delta
[r-prio6]     iPrio(l1, x, Skip)  = x   when
                  is-atom(x)            = true
[r-prio7]     iPrio(l1, Skip, x)  = Skip
[r-prio8]     iPrio(l1, x, y ; z) = iPrio(l1, x, y)
[r-prio9]     iPrio(l1, x, y + z) = iPrio(l1, iPrio(l1, x, y), z)
[r-prio10]    iPrio(l1, x ; y, z) = iPrio(l1, x, z) ; iPrio(l1, y, Delta)
[r-prio11]    iPrio(l1, x + y, z) = iPrio(l1, iPrio(l1, x, y), z) + iPrio(l1, iPrio(l1, y, x), z)
end Rewrite
```

```
process module Simulator
begin
        exports
        begin
                processes
                        Start     : Process
        end
        imports
                Rewrite
        atoms
                A         : Process
                print     : Process
                Skip      : Process
        processes
                P         : Process
        variables
                x, y      :         -> Process
        definitions
                Start(x)            = P(expand(process-id(x)))
                P(atom(x))          = A(x)
                P(x + y)            = P(expand(x)) + P(expand(y))
                P(x ; y)            = P(expand(x)) . P(expand(y))
                P(Skip)             = skip
end Simulator
```