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 extensions 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 \textit{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 \textit{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 \textit{sums} and \textit{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

\[ \text{process-def}(X) = a ; \text{process-id}(X) \]

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

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:

```plaintext
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

\[
P(\text{process-id}(x)) = P(\text{process-def}(x))
\]

\[
P(x) = A(x)
\]

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

```plaintext
functions
expand : Process -> Process
Delta : -> Process
```

```plaintext
equations
[e1] expand(\text{process-id}(x)) = expand(\text{process-def}(x))
[e2] expand(\text{process-def}(x)) = Delta
```

and changing the definitions to

\[
P(\text{process-id}(x)) = P(\text{expand}(\text{process-id}(x)))
\]

\[
P(\text{expand}(x)) = A(x)
\]

Note that we rely on the term-rewriting technique. We expect an innermost strategy, so that \(\text{process-def}(x)\) is rewritten before \(\text{expand}\) 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

\[
\text{process-def}(X) = a ; \text{process-id}(X)
\]

We simulate process \(X\) by

\[
P(\text{process-id}(X))
\]

This gives

\[
P(\text{expand}(\text{process-id}(X)))
\]

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

\[
P(\text{expand}(a ; \text{process-id}(X)))
\]

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

\[
P(a ; \text{process-id}(X))
\]
What matches the definition given in section 2.2, and thus results in

\[ P(\text{expand}(a)) \cdot P(\text{expand}(\text{process-id}(X))) \]

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.

```plaintext
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
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

\texttt{_:_:} : Process \# Process \to Process

To the module Expand we add:

equations

\[[e3]\] \quad \text{expand}(x : y) = x : y

And to the module Simulator:

definitions

\[ P(x : y) = P(\text{expand}(x)) \cdot P(\text{expand}(y)) \]

\subsection*{2.3 The + operator}

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

\texttt{_+_:} : Process \# Process \to Process

To the module expand we add the equation

\[[e4]\] \quad \text{expand}(x + y) = x + y

And to the module Simulator we add the definition

\[ P(x + y) = P(\text{expand}(x)) + P(\text{expand}(y)) \]

\subsection*{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.

\texttt{ Skip } : \to Process

To the module Expand we add

\[[e10]\] \quad \text{expand}(\text{Skip}) = \text{Skip}

and too the module Simulator

definitions

\[ P(\text{Skip}) = \text{skip} \]

\subsection*{2.5 The || operator}

We could easily do the following.

\texttt{_!!_:} : Process \# Process \to Process

\[[e5]\] \quad \text{expand}(x !! y) = x !! y

\[ P(x !! y) = P(\text{expand}(x)) \parallel P(\text{expand}(y)) \]

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, \textit{encaps}, and \textit{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.

\[ x \parallel y = x \parallel y + y \parallel x + x \parallel y \]

This implies that we have to do some rewriting in order to get terms without \( \parallel \) and \(|\).

To the module Operators we add the functions

\[ \_!\_ : \text{Process} \, \# \, \text{Process} \to \text{Process} \]
\[ \_!-\_ : \text{Process} \, \# \, \text{Process} \to \text{Process} -- \text{leftmerge} \]
\[ \_!\_ : \text{Process} \, \# \, \text{Process} \to \text{Process} -- \text{communication} \]

And to the module Expand

functions
\[ \_&!!_ : \text{Process} \, \# \, \text{Process} \to \text{Process} \]
equations
\[ \text{[e5]} \quad \text{expand}(x \&!! y) = \text{expand}(x &!! y) \]

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
\[ \text{[r-mrg]} \quad x \&!! y = (x \!- y) + (y \!- x) + \text{communicate}(x \! y) \]
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 somehow 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
end
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
end

atomic(x !! y) = false
atomic(x + y) = false
atomic(x ; y) = false
atomic(Skip) = false
atomic(Delta) = false
not(atomic(x)) = false
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
end

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 \texttt{!}- operator. Therefore we could add the following equations to module \texttt{Expand}.

\begin{verbatim}
\begin{align*}
\text{[lmerge1]} & \quad \text{process-id}(x) !- y = \text{process-def}(x) !- y \\
\text{[lmerge2]} & \quad (x \text{!!} y) !- z = (x \&\& y) !- z \\
\text{[lmerge3]} & \quad (\text{process-id}(x) ; y) !- z = (\text{process-def}(x) ; y) !- z
\end{align*}
\end{verbatim}

We need an equation for every possible term that has to be rewritten, when it appears as the left-hand-side of the \texttt{!}- 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.

\begin{verbatim}
data module Termtype
begin
  exports
  begin
    functions
      rewrite-term : Process \rightarrow \text{Boolean}
  end
end
imports
  Operators, Booleans
variables
  x, y : \rightarrow Process
equations
  \text{rew1} \quad \text{rewrite-term}(\text{process-id}(x)) = \text{true}
  \text{rew2} \quad \text{rewrite-term}(x \text{!!} y) = \text{true}
\end{verbatim}

And add the following equation to module \texttt{Expand}.

\begin{verbatim}
\begin{align*}
\text{[lmerge1]} & \quad x !- y = \text{expand}(x) !- y \quad \text{when} \quad \text{rewrite-term}(x) = \text{true} \\
\text{[lmerge2]} & \quad (x ; y) !- z = (\text{expand}(x) ; y) !- z \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}
\end{align*}
\end{verbatim}

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

\begin{verbatim}
\text{[atomic1]} \quad \text{atomic}(x) = \text{false} \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}
\end{verbatim}

And import the module \texttt{Termtype} instead of the modules Booleans and Operators.

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

\begin{verbatim}
functions
atom : Process \rightarrow Process
end
\end{verbatim}

To the module \texttt{Termtype}

\begin{verbatim}
equations
  \text{rew0} \quad \text{rewrite-term}(\text{atom}(x)) = \text{true}
\end{verbatim}

To the module \texttt{Expand}

\begin{verbatim}
equations
  \text{es1} \quad \text{expand}(x) = \text{atom}(x) \quad \text{when} \quad \text{is-atom}(x) = \text{true}
  \text{es2} \quad \text{expand}(\text{atom}(x)) = x
\end{verbatim}

And change the definition

\begin{verbatim}
P膨胀\texttt{(x)} = \texttt{A}(\texttt{x})
\end{verbatim}
in the module \texttt{Simulator} in
\[ P(\text{atom}(x)) = A(x) \]

Now, when \( \text{atom} \) is the result of an expansion, it appears as \( \text{atom}(a) \). When it is a sub-term, 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:

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

And module Rewrite

```plaintext
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.

```plaintext
[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
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 \( \text{element-of}(H, b) \) holds, the following rewritings are done.

\[
\begin{align*}
\text{element-of}(H, b) \\
\text{element-of}(\text{cons}(el(a), \text{cons}(el(b), \text{cons}(el(c), NIL))), b) \\
\text{element-of}(\text{cons}(el(a), \text{cons}(el(b), \text{cons}(el(c), NIL))), b) = \text{true} \\
\text{element-of}(\text{cons}(el(b), \text{cons}(el(c), NIL))), b) = \text{true} \\
\text{true} = \text{true}
\end{align*}
\]

The term \( \text{not}(\text{element-of}(H, c)) \) is rewritten to \( \text{false} \), and \( \text{not}(\text{element-of}(H, d)) \) to \( \text{true} \). 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 \times Set \to Set
  \_\_\_ : Set \times Set \to Set
  \_\_ : Set \times Set \to Set
equations

\[ \text{set4} \quad \text{element-of}(l1 + l2, x) = \text{true} \quad \text{when} \]
\[ \text{element-of}(l1, x) = \text{true} \]

\[ \text{set5} \quad \text{element-of}(l1 + l2, x) = \text{true} \quad \text{when} \]
\[ \text{element-of}(l2, x) = \text{true} \]

\[ \text{set6} \quad \text{element-of}(l1 \; l2) = \text{true} \quad \text{when} \]
\[ \text{element-of}(l1, x) = \text{true}, \]
\[ \text{element-of}(l2, x) = \text{true} \]

\[ \text{set7} \quad \text{element-of}(l1 \; \backslash l2) = \text{true} \quad \text{when} \]
\[ \text{element-of}(l1, x) = \text{true}, \]
\[ \text{not(element-of}(l2, x)) = \text{true} \]

Only the placeholder construction, as in
\[ H = \{ f(t) \mid t \in S \} \]
has not been specified yet. This gives a problem, because it introduces a new variable, which is something we cannot do in an 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.

\[ \text{set8} \quad \text{element-of}(\text{placeholder}(l1), x) = \text{true} \quad \text{when} \]
\[ \text{placeholder-match}(l1, x) = \text{true} \]

We can now specify a placeholder by doing the following.

functions
  PH1 : \to Set
equations

\[ \text{PH1} \quad \text{placeholder-match}(PH1, f(t)) = \text{true} \]

And using \text{placeholder}(PH1) in the construction of a set.

So we add the following to the module Sets.

functions
  placeholder : Set \to Set
  placeholder-match : Set \times Process \to \text{Boolean}
equations

\[ \text{set8} \quad \text{element-of}(\text{placeholder}(l1), x) = \text{true} \quad \text{when} \]
\[ \text{placeholder-match}(l1, x) = \text{true} \]

2.7 The \textit{encaps} operator

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

functions
  Encaps : Set \times Process \to Process
To the module Termtype
variables
  l : \to Set
equations

\[ \text{rew3} \quad \text{rewrite-term}(\text{Encaps}(l, x)) = \text{true} \]

And to the module Expand
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
variables
l1 : -> Set

equations
[r-hid1] iHide(l1, x) = Skip when
is-atom(x) = true,
not(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.

\[ \text{sum}(v \text{ in } S, P(v)) \]

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
\begin{align*}
\text{Sum} & : \text{Set \# Process} \rightarrow \text{Process} \\
\text{In} & : \text{Set \# Set} \rightarrow \text{Set}
\end{align*}

We want to specify the definition

\[ X = \text{sum}(x \text{ in } H, Y(x)) \]

in data.

sorts
\[ D \]

functions
\begin{align*}
Y & : D \rightarrow \text{Process} \\
X & : \rightarrow \text{Process} \\
a, b, c & : \rightarrow D \\
el & : D \rightarrow \text{Set}
\end{align*}

variables
\[ x : \rightarrow D \]

equations
[set] \[ H = \text{cons}(el(a), \text{cons}(el(b), \text{cons}(el(c), \text{NIL}))) \]
[def] \[ \text{process-def}(X) = \text{Sum}(\text{In}(el(x), H), Y(x)) \]

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
2.11 Extra rewrite rules

To the module Rewrite we add

\[ \begin{align*}
[r\text{-delta1}] & \quad x + \text{Delta} = x \\
[r\text{-delta2}] & \quad \text{Delta} + x = x \\
[r\text{-delta3}] & \quad \text{Delta} ; x = \text{Delta} \\
[r\text{-skip1}] & \quad x ; \text{Skip} = x \\
[r\text{-skip2}] & \quad (\text{Skip} ; x) + x = \text{Skip} ; x \\
[r\text{-skip1'}] & \quad x ; (\text{Skip} ; y) = x ; y
\end{align*} \]

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:

\[ P(x, y) = a(x) \cdot \text{if } x = y \text{ then } b(x) \text{ else } c(x) \]

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

\[ \text{If}(x, y, P) = P \text{ when } x = y \]

or

\[ \text{If}(x, x, P) = P \]

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.

\[ \text{equal}(x, x) = \text{true} \]
\[ \text{not}(\text{equal}(x, y)) = \text{true} \]

And now we can add the equation:

\[ \text{If}(x, y, P) = \text{Delta} \text{ when } \text{not}(\text{equal}(x, y)) = \text{true} \]

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:

\[ \text{prio}(\{a, b\}, a + b, c) \]

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 \( \text{prio}(\{a\}, a + b) \) equals \( a \), and \( \text{prio}(\{c\}, a + b) \) equals \( a + b \).

For manipulating with this new operator, we introduce a second one \( \text{iprio} \), 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 \( \text{skip} \), \( x \), \( y \), and \( z \) are process-expressions, and \( S \) is a set of atomic actions of which \( \text{skip} \) is not an element.

\[
\begin{align*}
\text{PRI1} & \quad \text{prio}(S, x) = \text{iprio}(S, x, \delta) \\
\text{PRI2} & \quad \text{iprio}(S, a, b) = a \quad \text{if } a \in S \lor b \notin S \\
\text{PRI3} & \quad \text{iprio}(S, a, b) = \delta \quad \text{otherwise} \\
\text{PRI4} & \quad \text{iprio}(S, a, \delta) = a \\
\text{PRI5} & \quad \text{iprio}(S, \delta, a) = \delta \\
\text{PRI6} & \quad \text{iprio}(S, a, \text{skip}) = a \\
\text{PRI7} & \quad \text{iprio}(S, \text{skip}, a) = \text{skip} \\
\text{PRI8} & \quad \text{iprio}(S, x, y, z) = \text{iprio}(S, x, y) \\
\text{PRI9} & \quad \text{iprio}(S, x, y + z) = \text{iprio}(S, \text{iprio}(S, x, y), z) \\
\text{PRI10} & \quad \text{iprio}(S, x, y + z) = \text{iprio}(S, x, z) \cdot \text{iprio}(S, y, \delta) \\
\text{PRI11} & \quad \text{iprio}(S, x + y, z) = \text{iprio}(S, \text{iprio}(S, x, y), z) + \text{iprio}(S, \text{iprio}(S, y, x), z)
\end{align*}
\]

This can easily be specified as follows.

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

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


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 Termttype
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(l, 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 Termttype
data module Atomic
begin
exports
begin
functions
begin
is-atom : Process -> Boolean
end
end
imports
Termttype
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
end

data module Expand
begin
exports
begin
functions
begin
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
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(atomic(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))
\[ e^8 \] \quad \text{expand}(\text{Sum}(l, x)) = \text{expand}(\text{iSum}(l, x))

\[ e^9 \] \quad \text{expand}(\text{Merge}(l, x)) = \text{expand}(\text{iMerge}(l, x))

\[ e^{10} \] \quad \text{expand}(\text{Skip}) = \text{Skip}

\[ e^{11} \] \quad \text{expand}(\text{Delta}) = \text{Delta}

\[ e^{12} \] \quad \text{expand}(\text{If}(d_1, d_2, x)) = \text{expand}(\text{if-else}(d_1, d_2, x, \text{Delta}))

\[ e^{13} \] \quad \text{expand}(\text{If-else}(d_1, d_2, x, y)) = \text{expand}(\text{if-else}(d_1, d_2, x, y))

\[ e^{14} \] \quad \text{expand}(\text{Prio}(l, x)) = \text{expand}(\text{iPrio}(l, x, \text{Delta}))

\[ \text{lmerge}^1 \] \quad x !- y = \text{expand}(x) !- y \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{lmerge}^2 \] \quad (x ; y) !- z = (\text{expand}(x) ; y) !- z \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{comm}^1 \] \quad x ! y = \text{expand}(x) ! y \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{comm}^2 \] \quad x ! y = x ! \text{expand}(y) \quad \text{when} \quad \text{rewrite-term}(y) = \text{true}

\[ \text{comm}^3 \] \quad (x ; y) ! z = (\text{expand}(x) ; y) ! z \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{comm}^4 \] \quad x ! (y ; z) = x ! (\text{expand}(y) ; z) \quad \text{when} \quad \text{rewrite-term}(y) = \text{true}

\[ \text{enc} \] \quad \text{iEncaps}(l, x) = \text{iEncaps}(l, \text{expand}(x)) \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{hid} \] \quad \text{iHide}(l, x) = \text{iHide}(l, \text{expand}(x)) \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{prio}^1 \] \quad \text{iPrio}(l, x, y) = \text{iPrio}(l, \text{expand}(x), y) \quad \text{when} \quad \text{rewrite-term}(x) = \text{true}

\[ \text{prio}^2 \] \quad \text{iPrio}(l, x, y) = \text{iPrio}(l, x, \text{expand}(y)) \quad \text{when} \quad \text{rewrite-term}(y) = \text{true}

\text{end Expand}
data module Sets begin
	exports begin
		functions
		emlement-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
	11, 12 : -> Set

equations
[set1] element-of(cons(el(x), 11), x) = true
[set2] element-of(cons(11, 12), x) = true when element-of(12, x) = true
[set3] not(element-of(11, x)) = true
[set4] element-of(11 + l2, x) = true when element-of(11, x) = true
[set5] element-of(11 + 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(11), x) = true when placeholder-match(11, x) = true
[set9] element-of(11, Skip) = false
[set10] element-of(11, Delta) = false
end Sets
data module Rewrite
begin
imports
Expand,
Sets
variables
w, x, y, z: -> Process
11, 12, 13: -> 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,
[r-cmm1b] is-atom(z) = true
[r-cmm2] (x; y) ! (z; w) = communicate(x ! z) ; (y !! w) when is-atom(x) = true,
[r-cmm3a] is-atom(z) = true
[r-cmm3b] is-atom(w) = true
[r-cmm4a] Delta ! x = Delta
[r-cmm4b] Delta ! Delta = Delta
-- encaps --
[r-enc1] iEncaps(l1, x) = Delta when is-atom(x) = true,
[r-enc2] element-of(l1, x) = true
[r-enc3] iEncaps(l1, x) = x when is-atom(x) = true,
[r-enc4] not(element-of(l1, x)) = true
[r-enc5] iEncaps(l1, x + y) = Encaps(l1, x) + Encaps(l1, y)
[r-enc6] iEncaps(l1, x ; y) = Encaps(l1, x) ; Encaps(l1, y)
-- hide --
[r-hid1] iHide(l1, x) = Skip when is-atom(x) = true,
[r-hid2] element-of(l1, x) = true
[r-hid3] iHide(l1, x + y) = iHide(l1, x) + iHide(l1, y)
[r-hid4] iHide(l1, x ; y) = iHide(l1, x) ; iHide(l1, y)
-- deadlock --
\[ \text{[r-delta1]} \quad x + \text{Delta} = x \]
\[ \text{[r-delta2]} \quad \text{Delta} + x = x \]
\[ \text{[r-delta3]} \quad \text{Delta} ; x = \text{Delta} \]

-- skip --
\[ \text{[r-skip1]} \quad \text{x ; Skip} = x \]
\[ \text{[r-skip2]} \quad (\text{Skip} ; x) + x = \text{Skip} ; x \]
\[ \text{[r-skip1']} \quad \text{x ; (Skip ; y)} = x ; y \]

-- sum --
\[ \text{[r-sum1]} \quad \text{iSum(cons(11, NIL), x)} = \text{fill-in}(11, x) \]
\[ \text{[r-sum2]} \quad \text{iSum(cons(11, cons(12, 13)), x)} = \text{fill-in} (11, x) + \text{Sum(cons(12, 13), x)} \]
\[ \text{[r-sum3]} \quad \text{iSum(NIL, x)} = \text{Delta} \]

-- merge --
\[ \text{[r-mrg1]} \quad \text{iMerge(cons(11, NIL), x)} = \text{fill-in}(11, x) \]
\[ \text{[r-mrg2]} \quad \text{iMerge(cons(11, cons(12, 13)), x)} = \text{fill-in} (11, x) !! \text{Merge(cons(12, 13), x)} \]
\[ \text{[r-mrg3]} \quad \text{iMerge(NIL, x)} = \text{Delta} \]
\[ \text{[r-comm1]} \quad \text{communicate(x ! y)} = \text{Delta} \]
\[ \text{[r-comm2]} \quad \text{communicate(x)} = x \quad \text{when} \]
\[ \text{[r-comm3]} \quad \text{communicate(Delta)} = \text{Delta} \]
\[ \text{[r-comm4]} \quad \text{communicate(Skip)} = \text{Skip} \]

-- if --
\[ \text{[r-if1]} \quad \text{iIf-else(d1, d1, x, y)} = \text{expand(x)} \]
\[ \text{[r-if2]} \quad \text{iIf-else(d1, d2, x, y)} = \text{expand(y)} \quad \text{when} \]
\[ \text{[r-prio1]} \quad \text{iPrio(l1, x, y)} = x \quad \text{when} \]
\[ \text{[r-prio2]} \quad \text{iPrio(l1, x, y)} = x \quad \text{when} \]
\[ \text{[r-prio3]} \quad \text{iPrio(l1, x, y)} = \text{Delta} \quad \text{when} \]
\[ \text{[r-prio4]} \quad \text{iPrio(l1, x, Delta)} = x \quad \text{when} \]
\[ \text{[r-prio5]} \quad \text{iPrio(l1, Delta, x)} = \text{Delta} \]
\[ \text{[r-prio6]} \quad \text{iPrio(l1, x, Skip)} = x \quad \text{when} \]
\[ \text{[r-prio7]} \quad \text{iPrio(l1, Skip, x)} = \text{Skip} \]
\[ \text{[r-prio8]} \quad \text{iPrio(l1, x, y ; z)} = \text{iPrio(l1, x, y)} \]
\[ \text{[r-prio9]} \quad \text{iPrio(l1, x, y + z)} = \text{iPrio(l1, iPrio(l1, x, y), z)} \]
\[ \text{[r-prio10]} \quad \text{iPrio(l1, x ; y, z)} = \text{iPrio(l1, x, z) ; iPrio(l1, y, Delta)} \]
\[ \text{[r-prio11]} \quad \text{iPrio(l1, x + y, z)} = \text{iPrio(l1, iPrio(l1, x, y), z) + iPrio(l1, iPrio(l1, y, x), z)} \]

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