

An Introduction to PSF_d

S. Mauw & G.J. Veltink

University of Amsterdam
Programming Research Group
P.O. Box 41882
1009 DB Amsterdam

abstract: PSF_d (Process Specification Formalism - Draft) is a Formal Description Technique developed for specifying concurrent systems. PSF_d supports the modular construction of specifications and parameterization of modules. As semantics for PSF_d a combination of initial algebra semantics and operational semantics for concurrent processes is used. This report is intended to give a brief introduction to the use of PSF_d.

Note: This work was sponsored in part by ESPRIT contract nr. 432, Meteor.

1. INTRODUCTION

PSF_d (Process Specification Formalism - Draft) has been designed as the base for a set of tools to support ACP (Algebra of Communicating Processes) [BK86b] and its formal definition can be found in [MV88]. ACP is a member of the family of concurrency theories, informally known as process algebras, and has already been applied to a large domain of problems, including: communication protocols [BK86a,Vaa86], algorithms for systolic systems [Weij87], electronic circuits [BV88] and CIM architectures [Mau87]. The size of these specifications is rather small such that manual verification can be achieved, but for industrially relevant problems we feel the need for a set of computer tools to help with the specification, simulation, verification and implementation.

Specifications in ACP, however, are written in an informal syntax and the treatment of data types is unspecified. The main goal in the design of PSF_d was to provide a specification language with a formal syntax, that would yet resemble ACP as much as possible, and to use a formal notion of data types. We have incorporated ASF (Algebraic Specification Formalism) [BHK87], which is based on the formal theory of abstract data types, in PSF_d to be able to specify data types by means of equational specifications. In order to meet the modern requirements of software engineering, like reusability of software, PSF_d provides the modular construction of specifications and parameterization of modules. This paper is meant to be an informal introduction to PSF_d. Please refer to [MV88] for more details.

The layout of this paper is as follows. In section 2 we show how data types are specified. Section 3 deals with the introduction of all operators used in defining the behaviour of processes. Along with the syntax the semantics of each operator is given. As a running example we will give the specification of a vending machine. This specification is adopted each time new language constructs are introduced. Modularization is the subject of section 4, in which import and export of data types and processes is treated. Section 5 gives the

specification of a Universal Vending Machine to illustrate the use of parameterization. An overview of the semantical issues is given in section 6. The last two sections give a comparison between PSF_d and LOTOS and a survey of the tools based on PSF_d.

2. DATA TYPES

A PSF specification consists of series of modules. There are two kinds of modules viz. data modules and process modules. In this section we deal with the data modules.

The first step in defining a data type is to define some *sorts* and some *functions* that operate on these sorts. The declaration of each function includes its *input-type* consisting of a list of zero or more sorts and its *output-type* consisting of exactly one sort. Functions that do not have an input-type, like the first two functions in the example, are called *constants*. The combination of sorts and functions is called the *signature* of a data type. Next we give an example of a simple definition and point out its constituents.

```
data module Booleans
begin
  sorts
    BOOLEAN

  functions
    true  :                               -> BOOLEAN
    false :                               -> BOOLEAN
    and   : BOOLEAN # BOOLEAN -> BOOLEAN
    or    : BOOLEAN # BOOLEAN -> BOOLEAN
    not   : BOOLEAN           -> BOOLEAN

  variables
    x,y : -> BOOLEAN

  equations
    [B1] and(true,x)  = x
    [B2] and(false,x) = false
    [B3] or(true,x)   = true
    [B4] or(false,x)  = x
    [B5] not(true)    = false
    [B6] not(false)   = true

end Booleans
```

This is an example of the definition of the data type *booleans*. The module is enclosed by two lines that state that the name of this *data module* is *Booleans*. There is one sort declared in this module called *BOOLEAN* and five functions among which two constants.

The signature of a data type gives all the information needed to construct well formed *terms*, which represent data values of that particular data type. Terms are constructed by applying an *n*-ary function to *n* terms of the correct type. This means that a constant, being a 0-ary function, is a term in itself. An example of a term generated by the signature of *booleans* is: *and(not(true), or(false,false))*. We are able to construct a lot of syntactically different terms, some of which might denote the same value. To state that two terms

denote the same value we use *equations*. An example of such an equation is: $\text{and}(\text{true}, \text{false}) = \text{false}$. More generally we could say that for every boolean term x , the equation $\text{and}(\text{true}, x) = x$ holds. In this case x is a variable of the sort *BOOLEAN*. See the example for the complete list of equations that we stated to hold for the booleans.

As the semantics for the data types we use the *initial algebra* semantics as defined in [EM85, GM85]. In short this means that all terms that are equal, as derivable from the equations, are in the same equivalence class. Each equivalence class corresponds with exactly one element of the initial algebra. We write $[t]$ for the equivalence class of a term t .

3. PROCESSES

In this and the following sections we focus on the process modules. Processes in PSF_d are described as a series of *atomic actions* combined by operators. Atomic actions are the basic and indivisible elements of processes in PSF_d . 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. From now on we will introduce the operators one by one, but first we will have to introduce the *action rules*, i.e. the notation we use to express the semantics of an expression. Action rules were introduced by Plotkin in [Plo82] to give an operational semantics for CSP [Hoa85].

For each atomic action a we define a binary relation $\bullet \xrightarrow{a} \bullet$ and a unary relation $\bullet \xrightarrow{a} \checkmark$ on closed process expressions, i.e. process expressions containing no variables. The notation $x \xrightarrow{a} y$ means that a process expression represented by x can evolve into y by executing the atomic action a and $x \xrightarrow{a} \checkmark$ means that the process expression represented by x can terminate successfully after having executed the atomic action a . The special symbol \checkmark can be looked upon as a symbol indicating successful termination of a process. When using action relations in this document the a always stands for an atomic action and the x and y stand for a process expression. Beware that in this document we do not give the complete list of action rules because it is meant as an introduction.

We start with an axiom that states that a process expression consisting of an atomic action a only, can terminate successfully by executing atomic action a . This fact is expressed by the following action rule:

$$a \xrightarrow{a} \checkmark$$

Sequential composition is expressed by using the \cdot -operator like in: $a \cdot b$, which states that after atomic action a has been executed, atomic action b can be executed. The semantics for sequential composition are given by:

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \quad \frac{x \xrightarrow{a} \checkmark}{x \cdot y \xrightarrow{a} y}$$

The second rule, e.g., states that whenever a process expression x can terminate execution action a , the process expression $x.y$ is able to execute action a and to evolve into process expression y .

Alternative composition is expressed by using the '+'-operator like in: $a + b$, which states that a non-deterministic choice is made between a and b first and that the chosen action is executed after that.

The semantics for alternative composition are given by:

$$\frac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'} \quad \frac{x \xrightarrow{a} \checkmark}{x+y \xrightarrow{a} \checkmark} \quad \frac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'} \quad \frac{y \xrightarrow{a} \checkmark}{x+y \xrightarrow{a} \checkmark}$$

With these simple operations we are already able to specify a simple vending machine. Our vending machine sells coffee for 25 cents and tea for 10 cents.

```

process module Vending-Machine
begin

  atoms
    10c, 25c, coffee, tea

  processes
    VCT

  definitions
    VCT = ((10c . tea) + (25c . coffee)) . VCT

end Vending-Machine

```

There are some new features that appear in this example. The atomic actions are introduced in the *atoms* section. In the *processes* section the names for processes are declared, while the behaviour of a process is defined in the *definitions* section. In the definition of VCT we see that after delivering a cup of tea or a cup of coffee the machine returns to its original state, which is expressed by repeating the name of the process at the end of the right-hand side of the equation. This feature is called *recursion*.

We give the initial part of a possible trace, i.e. a series of derivations, of this vending machine. In this trace we will leave out the intermediate processes because we are only interested in the atomic actions that occur.

$$VCT \xrightarrow{10c} \dots \xrightarrow{tea} \dots \xrightarrow{25c} \dots \xrightarrow{coffee} \dots \xrightarrow{25c} \dots \xrightarrow{coffee} VCT$$

Next we want to introduce parallel composition, which is expressed by using the '||'-operator. The expression $x || y$ states that the processes x and y are executed in parallel. To execute in parallel means that the first atomic action executed by $x || y$ may come from

either x or y , or that the first atomic actions from both x and y can communicate with each other. This is called interleaving concurrency. The expression $a|b = c$ states that two atomic actions a, b can communicate and that the result will be another atomic action c . The semantics for parallel composition are given by:

$$\begin{array}{c}
 \frac{x \xrightarrow{a} x'}{x||y \xrightarrow{a} x'||y} \quad \frac{x \xrightarrow{a} \checkmark}{x||y \xrightarrow{a} y} \quad \frac{y \xrightarrow{a} y'}{x||y \xrightarrow{a} x||y'} \quad \frac{y \xrightarrow{a} \checkmark}{x||y \xrightarrow{a} x} \\
 \\
 \frac{x \xrightarrow{a} x'; y \xrightarrow{b} y'; a|b=c}{x||y \xrightarrow{c} x'||y'} \quad \frac{x \xrightarrow{a} \checkmark; y \xrightarrow{b} \checkmark; a|b=c}{x||y \xrightarrow{c} \checkmark} \\
 \\
 \frac{x \xrightarrow{a} x'; y \xrightarrow{b} \checkmark; a|b=c}{x||y \xrightarrow{c} x'} \quad \frac{x \xrightarrow{a} \checkmark; y \xrightarrow{b} y'; a|b=c}{x||y \xrightarrow{c} y'}
 \end{array}$$

Suppose we want to add some users to the specification. In this example we will model a situation in which a client that likes to have tea arrives at the vending machine followed by a client that wants coffee.

```

process module Vending-Machine-and-Users
begin
  atoms
    insert-10c, accept-10c, 10c-paid,
    insert-25c, accept-25c, 25c-paid,
    serve-coffee, take-coffee, coffee-delivered,
    serve-tea, take-tea, tea-delivered

  processes
    VMCT, Tea-User, Coffee-User, System

  sets
    of atoms
      H = { insert-10c, accept-10c, insert-25c, accept-25c,
            serve-coffee, take-coffee, serve-tea, take-tea }

  communications
    insert-10c | accept-10c = 10c-paid
    insert-25c | accept-25c = 25c-paid
    serve-tea  | take-tea  = tea-delivered
    serve-coffee | take-coffee = coffee-delivered

  definitions
    VMCT = ((accept-10c . serve-tea) +
            (accept-25c . serve-coffee)) . VMCT
    Tea-User = insert-10c . take-tea
    Coffee-User = insert-25c . take-coffee

    System = encaps (H, VMCT || (Tea-User . Coffee-User))

end Vending-Machine-and-Users

```

The specification has grown considerably. We will have a look at the new features that have been introduced. The first thing we notice is that the amount of atomic actions has

increased. This is due to the fact that we now have four pairs of communicating atomic actions. These pairs and their results are listed in the *communications* section. The next new feature is the *sets* section. It is possible in PSF_d to assign a name to a set of terms of a given sort, in this case the *predefined* sort *atoms*. In this example all atomic actions that are not the result of a communication are put in the set *H*. This set is used in the last line of the *definitions* section by the *encaps* (encapsulation) operator. The process expression *encaps*(*H*,*x*) is equal to the process expression *x* without the possibility of performing atomic actions from *H*. This construction is used to force communication between certain atomic actions.

The semantics of the *encaps* operator are given by:

$$\frac{x \xrightarrow{a} x'; a \in H}{\text{encaps}(H, x) \xrightarrow{a} \text{encaps}(H, x')} \quad \frac{x \xrightarrow{a} \checkmark; a \notin H}{\text{encaps}(H, x) \xrightarrow{a} \checkmark}$$

The only possible trace of this system is:

$$\text{System} \xrightarrow{10c\text{-paid}} \dots \xrightarrow{\text{tea-delivered}} \dots \xrightarrow{25c\text{-paid}} \dots \xrightarrow{\text{coffee-delivered}} \text{encaps}(H, \text{VMCT})$$

Now suppose we are not interested in the atomic actions that occur when the money has been paid. PSF_d offers the *hide* operator to rename all unwanted actions into *skip*. Its semantics are given by:

$$\frac{x \xrightarrow{a} x'; a \in I}{\text{hide}(I, x) \xrightarrow{\text{skip}} \text{hide}(I, x')} \quad \frac{x \xrightarrow{a} \checkmark; a \in I}{\text{hide}(I, x) \xrightarrow{\text{skip}} \checkmark}$$

$$\frac{x \xrightarrow{a} x'; a \notin I}{\text{hide}(I, x) \xrightarrow{a} \text{hide}(I, x')} \quad \frac{x \xrightarrow{a} \checkmark; a \notin I}{\text{hide}(I, x) \xrightarrow{a} \checkmark}$$

From these action relations for *hide* it is clear that *skip* can also act as a label of a transition, even though it is no atomic action.

To get rid of the unwanted actions in the previous example we define an extra set *I* in the *sets* section and change the definition of *System* in the *definitions* section to include the *hide* operator.

```
I = { 10c-paid, 25c-paid }
.....
System = hide(I, encaps(H, VMCT || (Tea-User . Coffee-User)))
```

The only possible trace of the system would now be:

$$\text{System} \xrightarrow{\text{skip}} \dots \xrightarrow{\text{tea-delivered}} \dots \xrightarrow{\text{skip}} \dots \xrightarrow{\text{coffee-delivered}} \text{encaps}(H, \text{VMCT})$$

4. MODULARIZATION

The next thing we want to do is to specify a system of a vending machine and clients in a modular fashion. The three sections in PSF_d that deal with modularity are the *exports*, *imports* and *parameters* section. All definitions that are listed in the *exports* section are visible outside the module. A data module may define sorts and functions, while a process module may define atoms, processes and sets in the *exports* section. All objects that are declared outside the *exports* section are called *hidden* and are only visible inside the module in which they were declared. When a module *A* imports a module *B*, all names in the *exports* section of *B* are automatically exported by *A* too. This feature is called *inheritance*.

To start our modular specification of the vending machine we define some amounts of money that it accepts.

```
data module Amounts
begin
  exports
  begin
    sorts
      AMOUNT
    functions
      10c : -> AMOUNT
      20c : -> AMOUNT
      25c : -> AMOUNT
      30c : -> AMOUNT
  end
end Amounts
```

The initial algebra of the sort *AMOUNT* in this module now consists of four elements namely: [10c], [20c], [25c], [30c].

The basic way to combine modules is by way of import. In the *imports* section we define which modules have to be imported. By importing module *A* in module *B*, all exported objects from *A* become visible inside *B*. It is not allowed to import a process module into a data module. Now we give a definition of some drinks and their prices. The module *Amounts* is imported as to be able to use the sort *AMOUNT*.

```
data module Drinks
begin
  exports
  begin
    sorts
      DRINK
    functions
      tea : -> DRINK
      coffee : -> DRINK
      orange : -> DRINK
      price : DRINK -> AMOUNT
  end
end
```



```

imports
  Amounts

equations
[P1] price(tea)      = 10c
[P2] price(coffee)   = 25c
[P3] price(orange)   = 30c

end Drinks

```

This module defines a sort *DRINK* containing three elements and a function *price* from *DRINK* to *AMOUNT*.

Next we define a client that has its own favourite drink.

```

process module Drinks-User
begin
  exports
    begin
      atoms
        select      : DRINK*
        insert      : AMOUNT
        take-drink  : DRINK
      processes
        user : DRINK
    end
  imports
    Drinks

  variables
    fav-drink : -> DRINK          -- the user's favourite drink

  definitions
    user(fav-drink) = select(fav-drink) .
                      insert(price(fav-drink)) .
                      take-drink(fav-drink)

end Drinks-User

```

In this example we see that *atoms* as well as *processes* can take data elements as parameters. The process *user* is parameterized by the user's favourite drink, see the line *user(fav-drink) = select(fav-drink)*. So now we have defined three users namely: *user(tea)*, *user(coffee)* and *user(orange)*. These processes all have the same behaviour, except for the drinks that are subject to their actions. So the first action of the process *user(tea)* is *select(tea)*, whereas the first action of process *user(coffee)* is *select(coffee)*.

5. PARAMETERIZATION

To be able to exploit the reusability of specifications, a parameterization concept is included in PSF_d. Parameterization is described in the *parameters* section and takes the form of a sequence of formal parameters. Each parameter is a block that has a name and lists some

formal objects. Parameters in a data module may consist of sorts and functions only, whereas parameters in a process module consist of atoms, processes and sets additionally. In the next example we define a universal vending machine that has the items it sells as a parameter. These items are represented by the sort *PRODUCT* and we demand that there is a function *price* from *PRODUCT* to *AMOUNT*.

```

process module Universal-Vending-Machine
begin
  parameters
    Items-on-sale
    begin
      sorts
        PRODUCT
      functions
        price : PRODUCT -> AMOUNT
    end Items-on-sale

  exports
    begin
      atoms
        get-selection : PRODUCT
        accept         : AMOUNT
        serve-product : PRODUCT
      processes
        UVM
    end

  imports
    Amounts

  variables
    chosen-item : -> PRODUCT

  definitions
    UVM = sum(chosen-item in PRODUCT,
              get-selection(chosen-item).
              accept(price(chosen-item)).
              serve-product(chosen-item)
            ) . UVM

end Universal-Vending-Machine

```

The intuitive idea behind the Universal Vending Machine is the following:

- for each product
 - offer the possibility to select this product
 - accept the amount of money to be paid for this product
 - serve the chosen product

In this example the *sum* operator, which acts as a generalization of the alternative composition (+), is introduced. A so-called *placeholder* (*chosen-item*) is used to define a process expression containing a kind of variable. The sum operator takes two arguments, the *placeholder definition* (*chosen-item in PRODUCT*), which defines the domain of the placeholder, and a process expression, to which the scope of this placeholder is limited. In

this example the *sum* operator introduces one process expression for each element of *PRODUCT*, as part of one big alternative composition.

There is another operator that resembles the *sum* operator, namely the *merge* operator that generalizes the parallel composition in a similar way. This operator will not be dealt with in this paper.

Whenever a parameterized module is imported into another module, each parameter of the former module may become bound to a third module by binding all objects listed in the parameter to actual sorts, functions, atoms, processes and sets from this third module. All unbound parameters are *inherited* by the importing module and are indistinguishable from the parameters defined in its own *parameters* section.

In the next example we make a specification of a vending machine and two users by using the modules we have already defined.

```

process module VM-Tea-Coffee-Orange
begin

  imports
    Universal-Vending-Machine
    { Items-on-sale
      bound by
        [PRODUCT -> DRINK]
      to Drinks
      renamed by
        [get-selection -> watch-button,
          UVM           -> VMCTO,
          serve-product -> serve-drink]],
    Drinks-User
    { renamed by
      [select -> push-button ]

  atoms
    order, delivered : DRINK
    paid : AMOUNT

  processes
    System

  sets
    of atoms
      H = { push-button(d), watch-button(d) | d in DRINK } +
          { serve-drink(d), take-drink(d) | d in DRINK } +
          { insert(c), accept(c) | change in AMOUNT }

  communications
    push-button(d) | watch-button(d) = order(d) for d in DRINK
    serve-drink(d) | take-drink(d)   = delivered(d) for d in DRINK
    insert(c)      | accept(c)       = paid(c) for c in AMOUNT

  definitions
    System = encaps (H, VMCTO || ( user(tea) . user(coffee) ))

end VM-Tea-Coffee-Orange

```

The visible names of a module can be renamed by the use of the *renamed by* construct, which specifies a renaming by giving a list of pairs of renamings in the form of an old

visible name and a new visible name. Thus we specify the interaction between the user and the vending machine in this example by means of buttons (*watch-button*, *push-button*).

The *bound by* construct is used to bind parameters and specifies the name of a parameterized module, a parameter name, a list of bindings (pairs consisting of a formal name and an actual name), and the name of an actual module. Thus we have bound the parameter *Items-on-sale* of the *UVM* to the module *Drinks*, obtaining a Tea-Coffee-Orange Vending Machine.

6. MORE ON SEMANTICS

In [MV88] the formal semantics of PSF_d are described. To shape the intuitive notion of semantics treated so far, we will elaborate on it in this section. To assign a semantics to a modular PSF_d specification we use a normalization procedure that removes all modular structure. It produces one *flat* data module and one flat process module which imports the flat data module. The following picture shows the several levels of semantics involved in the formal definition.

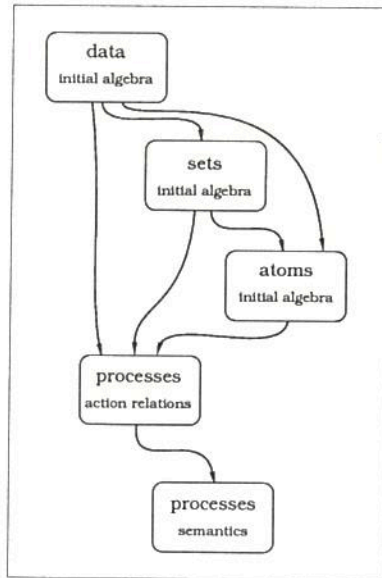


figure 1. Dependencies among different semantic domains.

The semantics of the data module is the initial algebra semantics as pointed out before. The semantics of the objects defined in the process module are based upon the initial algebra semantics of the data types. Sets can be understood as subsorts of a given sort. Atomic actions are defined using the predefined sort *atoms* and possibly take elements of the data types as parameters. There is an equivalence relation defined on the atomic actions, which is induced by the initial algebra semantics of the data types. We will illustrate this by giving

an example related to the module *Drinks-User* as defined in section four. Whenever a closed term occurs as a parameter of an atomic action, it should be looked upon as representing its equivalence class in the initial algebra. In fact we should have written $[t]$ for each data term t in the specification, but we leave out the brackets for reasons of simplicity. So because $price(orange)$ represents the same object as $30c$, the atomic action $insert(price(orange))$ is equal to $insert(30c)$.

In section 3 we have defined an operational semantics for process expressions by means of action relations. These action relations are suitable to define a semantic domain, i.e. the *graph model*, on which most of the known equivalence relations on processes can be defined. In this way we can assign a labeled directed transition graph to each process. We define *bisimulation* equivalence [Par81] on these graphs as the intended semantics for PSF_d processes.

7. COMPARISONS

Compared with other FDT's (Formal Description Techniques) PSF_d is most closely allied to LOTOS [ISO87]. LOTOS is one of the two FDT's developed within ISO (International Organization for Standardization) for the formal specification of open distributed systems. Like PSF_d , LOTOS is a combination of two formalisms, namely a variant of ACT ONE [EM85] to describe data types and a process description part based on CCS [Mil80]. One of the design goals of PSF_d was to stay as close to ACP as possible. The result of this goal is that the distance between PSF_d and ACP is much smaller than the distance between LOTOS and CCS.

The main differences between PSF_d and LOTOS originate from the differences between ACP and CCS. Sequential composition is expressed in CCS by means of the *action prefix* operator. This operator combines an action and a process or *behaviour expression*. To link two processes together one has to use another operator, the *enable* operator. In ACP atomic actions are looked upon as being elementary processes, therefore only one operator is needed to express sequential composition.

In LOTOS communication is established by synchronization of *observable actions* with the same name. In ACP the communication function is used to define which atomic actions are able to communicate. We think of this as an advantage when systems are specified in a modular fashion, because it gives the possibility to develop modules independently and tie them together by specifying the communication function afterwards. In contrast, in LOTOS the names of all gates have to be known in advance.

The data specification parts of PSF_d and LOTOS are very similar. This includes parameterization and renaming of imported sorts and functions. However it is not possible to define hidden signatures in LOTOS.

Though modularization is possible when defining data types, LOTOS does not support such a powerful concept of importing and exporting processes and actions as opposed to PSF_d , which supports one global concept of modularization. The only way to have some

abstraction in LOTOS is by writing a specification in a stringent top-down manner using the *where* construction, in which the subprocesses have to be specified explicitly each time. The next piece of a LOTOS specification from [BB87] will clarify this notion.

```

process Sender[ConReq, ConCnf, DatReq, DisReq] :=
  Connection-Phase[ConReq, ConCnf] » Data-Phase[DatReq, DisReq]

where
  process Connection-Phase[ConReq, ConCnf] :=
    ConReq; ConCnf; exit
  endproc
  process Data-Phase[DatReq, DisReq] :=
    (DatReq; Data-Phase[DatReq, DisReq]
    [] DisReq; stop)
  endproc
endproc

```

We claim that such an approach does not support the reusability of specifications and we think that it will lead to monolithic specifications that are harder to understand due to the lack of a proper abstraction mechanism.

We refer to [MV88] for a more extensive comparison between PSF_d and LOTOS as well as some other FDT's and programming languages.

8. TOOLS

As stated in the introduction, PSF_d has been designed as the base for a set of tools. The first tool we are currently implementing is a simulator. The goal is to come up with a program that is able to simulate, possibly in interaction with the user, the processes that are defined in the PSF_d specification. The first phase of this implementation, being a syntax and type checker, has already been accomplished. In constructing this simulator we hope we will gain more experience and ideas to build a verification tool, for testing equivalence of processes, and as the last step an implementation tool, that will implement a specification in some kind of programming language, hopefully to be executed on a parallel computer.

9. CONCLUSIONS

In this report we have presented PSF_d, a new formalism to describe process behaviour. We have shown that it is possible to integrate a formal approach towards data types in this formalism and as an example we gave the specification of a vending machine in PSF_d. PSF_d also has been used for specifications other than toy examples. We refer to [MV88] for a detailed specification of the Alternating Bit Protocol making full use of the modularization concepts, as well as some other more elaborate examples. We hope that PSF_d will be able to serve as a contribution to the construction of more reliable software.

10. REFERENCES

- [BB87] T. Bolognesi & E. Brinksma, *Introduction to the ISO Specification Language LOTOS*, in: Computer Networks and ISDN Systems 14, pp 25-59, North-Holland, Amsterdam, 1987.
- [BHK87] J.A. Bergstra, J. Heering & P. Klint, *ASF - An algebraic specification formalism*, Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, 1987.
- [BK86a] J.A. Bergstra & J.W. Klop, *Verification of an alternating bit protocol by means of process algebra*, in: Math. Methods of Spec. & Synthesis of Software Systems '85, (W. Bibel & K.P. Jantke, eds.), Math. Research 31, Akademie-Verlag Berlin, pp 9-23, 1986.
- [BK86b] J.A. Bergstra & J.W. Klop, *Process algebra: specification and verification in bisimulation semantics*, in: Math. & Comp. Sci. II, (M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens, eds.), CWI Monograph 4, pp 61-94, North-Holland, Amsterdam, 1986.
- [BV88] J.C.M. Baeten & F.W. Vaandrager, *Specification and Verification of a circuit in ACP*, Report P8803, University of Amsterdam, 1988.
- [EM85] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specifications, Vol. I, Equations and Initial Semantics*, Springer-Verlag, 1985.
- [GM85] J.A. Goguen & J. Meseguer, *Initiality, induction and computability*, in: Algebraic Methods in Semantics (M. Nivat & J.C. Reynolds eds.), pp. 460-541, Cambridge University Press, 1985.
- [Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
- [ISO87] International Organization for Standardization, *Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO/TC 97/SC 21, (E. Brinksma, ed.), 1987.
- [Mau87] S. Mauw, *Process Algebra as a Tool for the Specification and Verification of CIM-architectures*, Report P8708, University of Amsterdam, Amsterdam, 1987.
- [Mil80] R. Milner, *A calculus of communicating systems*, Springer LNCS 92, 1980.
- [MV88] S. Mauw & G.J. Veltink, *A Process Specification Formalism*, Report P8814, University of Amsterdam, Amsterdam, 1988.
- [Par81] D.M.R. Park, *Concurrency and automata on infinite sequences*, in: Proc. 5th GI Conf., (P. Deussen, ed.), Springer LNCS 104, pp 167-183, 1981.
- [Plo82] G.D. Plotkin, *An operational semantics for CSP*, in: Proc. Conf. Formal Description of Programming Concepts II, Garmisch 1982 (E. Björner, ed.), pp. 199-225, North-Holland, 1982.
- [Vaa86] F.W. Vaandrager, *Verification of two communication protocols by means of process algebra*, Report CS-R8606, Centre for Mathematics and Computer Science, Amsterdam, 1986.
- [Weij87] W.P. Weijland, *Correctness proofs for systolic algorithms: a palindrome recognizer*, Report CS-R8747, Centre for Mathematics and Computer Science, Amsterdam, 1987.
To appear in: Theoretical Foundations of VLSI design, (K. McEvoy & J.V. Tucker, eds.)