Fundamenta Informaticae XIII (1990) 85-139
IOS Press

A PROCESS SPECIFICATION FORMALISM †

S. MAUW and G.J. VELTINK

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

Traditional methods for programming sequential machines are inadequate for specifying parallel systems. Because debugging of parallel programs is hard, due to e.g. non-deterministic execution, verification of program correctness becomes an even more important issue. The Algebra of Communicating Processes (ACP) is a formal theory which emphasizes verification and can be applied to a large domain of problems ranging from electronic circuits to CAM architectures. The manual verification of specifications of small size has already been achieved, but this cannot easily be extended to the verification of larger industrially relevant systems. To deal with this problem we need computer tools to help with the specification, simulation, verification and implementation. The first requirement for building such a set of tools is a specification language. In this paper we introduce PSF_d (Process Specification Formalism - draft) which can be used to formally express processes in ACP. In order to meet the modern requirements of software engineering, like reusability of software, PSF_d supports the modular construction of specifications and parameterization of modules. To be able to deal with the notion of data, ASF (Algebraic Specification Formalism) is embedded in our formalism. As semantics for PSF_d a combination of initial algebra semantics and operational semantics for concurrent processes is used. A comparison with programming languages and other formal description techniques for the specification of concurrent systems is included.

1 MOTIVATION

The last decade has shown yet another revolution in computer technology: parallel computers. And with this progression in hardware ought to come a progression in the development of software. Though software development has come a long way, since the days that Alan Turing fed machine code programs written in Baudot code into the Colossus [Hod83], through assemblers, higher level languages like FORTRAN and later Pascal and even Ada, it still finds itself in the midst of the so-called *software crisis*. With the introduction of parallel computers things tend to become even worse.

† This work was partially supported by ESPRIT contract no. 432, An integrated formal approach to industrial software development (METEOR).

1.1 PROBLEMS WITH SOFTWARE

It seems that developers of software were not yet ready for this next step. It is a well-known fact that large programs, though being used in every day services, have the nasty property of still containing software errors or *bugs* as they are called in the vernacular. That programmers have become used to this fact show the UNIX reference manuals [BTL79], wherein, in the standard format for describing programs, there is a special section devoted to the known bugs. Construction of large programs is difficult, however, maintaining and tailoring programs to new needs is far more complex. In many cases it is better to write the whole program from scratch again than to try and adapt it.

1.2 BUILDING BETTER PROGRAMS

Of course there have been attempts to develop methods to help programmers in constructing software and excluding errors. One of the most formal approaches, using mathematical techniques, is the proving of program correctness. See for example the work by Dijkstra [Dijk75] and Hoare [Hoa72]. Though these formal methods do well for small programs, they have never really found their way to the programming-in-the-large. Other methods use data flow charts and all kinds of graphical representations of the program to help the programmer.

Yet another method to support program development is the use of programming environments. A programming environment consists of a large number of tools and an environment that help the programmer in constructing programs. Some examples of these tools, currently available, are (syntax-directed) editors, debuggers, compilation aids (like the make program on UNIX machines) and so on, but also hierarchical file systems and facilities to communicate with other programmers on the same computer, or even on a network. In designing a new programming language more and more attention is being paid to such an environment. See for example Ada [USD80], where a set of requirements has been defined that a programming environment should meet to be called an Ada Program Support Environment (APSE).

An example of an even more integrated system is the IOTA programming system [NY83] that offers a set of programs consisting of a syntax-directed editor, compiler, debugger and a correctness prover, that all work on a huge database in which program modules are being represented in some internal representation. In this way the cooperation between programmers and the reusability of software modules is highly improved.

1.3 PARALLEL PROGRAMMING

With the introduction of parallelism in programming two main approaches can be seen in the field of programming languages. On the one hand, existing programming languages have been extended to incorporate features that deal with parallelism like Concurrent Pascal [Bri75], on the other hand new languages have been developed that are suitable for writing parallel programs from the start, like OCCAM and Ada [Bar82]. We think that with the introduction of concurrency in programming languages, programmers have to start thinking of solving problems in a parallel way, as opposed to the sequential way of thinking imposed by the von Neumann computers. Therefore we are in favour of programming languages especially designed to support concurrency. Such programming languages will be the first step towards real parallel programming.

1.4 MATHEMATICAL CONCURRENCY THEORIES

ACP (Algebra of Communicating Processes) [BK86c], or more informal process algebra, is one of the many mathematical theories for concurrency. Other examples from this family of theories are: CSP, CCS, Petri nets, trace theory, temporal logic and denotational semantics. Specifications in ACP have been applied to a large domain of problems ranging from communication protocols [BK86a,Vaa86a], algorithms for systolic systems [Weij87] and electronic circuits [BV88] up to CIM architectures [Mau87b]. The manual verification of specifications of small size has already been achieved, but for industrially relevant problems we feel the need for a set of computer tools to help us with the specification, simulation, verification and implementation. These tools will together form a programming or *specification environment*. The first requirement for building such a set of tools is a specification language that is based on ACP.

1.5 MOTIVATION FOR PSF_d

A first attempt has been made in [Mau87a] to give an algebraic specification of ACP. In this report it is concluded that the transformation of process algebra into an algebraic specification is quite easy, but that the transformation of an application of process algebra into an algebraic specification takes more effort. It was also stated in this report, that in specifying process algebra applications in some formal language, one has to be more accurate with respect to the specification of the data types and the ports at which processes communicate. Another disadvantage of specifying process algebra applications by means of algebraic specifications is that the specifications do not have the same appearance as the ones we are used to.

To deal with these problems we have decided to start the development of a new specification formalism called PSF_d (Process Specification Formalism - draft) that would be based on two concepts. Firstly the specification of the data types must be performed in some algebraic specification formalism and secondly the specification of the behaviour of processes must be performed in some formalism especially designed for this need. Another example of such a project is LOTOS [ISO87] in which ACT ONE [EM85] has been combined with CCS [Mil80]. We have chosen ASF [BHK89] as the algebraic specification formalism and ACP as the base for the formalism as presented in this paper. For a short explanation of ASF, see section 2.3 and for a short explanation of ACP, see the following section. An introduction to PSF_d csn be found in [MV89].

2 DEFINITION & DESCRIPTION

In this section we will give a description of the formalisms that PSF_d is based on as well as the definition of its syntax.

2.1 EXPLANATION OF ACP

ACP is a theory that deals with concurrent, communicating processes. These processes can be the execution of an algorithm by a computer as well as the description of a drinks dispenser or actions of human beings. However the current interest focuses mainly on *distributed systems* and *communication protocols*.

2.1.1 General Setting

The development of ACP started in 1982, at the Centre for Mathematics and Computer Science in Amsterdam, by J.A. Bergstra and J.W. Klop. Compared with other concurrency theories, ACP is most closely allied to CCS. There is, however, one important difference; the starting point of CCS, like CSP and Petri nets, is some model of concurrency whereas the starting point of ACP is a system of axioms. In the first approach an algebraic structure is obtained by abstracting from certain aspects of processes. There are usually some basic objects or atoms and ways of constructing more complex expressions from these basic objects. Next, equivalences in this model are investigated and general rules are formulated.

In ACP, on the contrary, a set of rules is defined first. These rules hold in most models that have been proposed. This way we get a more common algebraic theory, such that whenever a new rule is introduced we can find out in which class of models it holds and in which it does not. We can try to describe a model just by rules and we can find out which operators can be defined in a model and which ones cannot. Because of this approach we are able to compare theories of concurrency and the pros and cons of, e.g., CSP and CCS can be discussed. See for example [Gla86].

2.1.2 An Introduction

In this section we will give a brief introduction to ACP. This introduction is by no means intended to be complete, but merely gives an intuitive notion of what we are dealing with. For a complete introduction to ACP we refer to [BK86c] and [Bae86].

ACP starts from a set of objects, called atomic actions, atoms or steps. Atomic actions are the basic and indivisible elements of ACP and will be represented in the sequel by the symbols a,b,c. In ACP all atoms are constants. Moreover, we have two extra constants:

- δ, deadlock.
  deadlock is the acknowledgement that there is no possibility to proceed.
- τ, silent action.
  τ represents the process terminating after some time, without performing observable actions.

Processes, which will be denoted by the symbols x,y, are generated from the constants by means of operators. A few examples of such operators are:

- $\cdot$, sequential composition or product.
  $x \cdot y$ is the process that executes x first and continues with y after termination of x.
- $+$, alternative composition or sum.
  $x+y$ is the process that first makes a choice between its summands x and y, and then proceeds with the execution of the chosen summand. In the presence of an alternative, δ is never chosen.
- $\|$, parallel composition or merge.
  $x \| y$ is the process that represents the simultaneous execution of x and y.
- ∂_H, encapsulation.
  $\partial_H(x)$ is the process x without the possibility of performing actions from the set of atomic actions H. Algebraically this is achieved by renaming all atomic actions from H in x into δ.

- τ_I, abstraction.

  $\tau_I(x)$ is the process x without the possibility of observing actions from the set of atomic actions I. This is achieved by renaming all atomic actions from I in x into τ.

There are many more operators and predicates on processes, but we will not present them here. As stated earlier ACP is capable of dealing with several models, generated by different sets of axioms. The simplest of this axiom systems is called Basic Process Algebra (BPA). Its axioms are:

1. $x + y = y + x$
2. $(x + y) + z = x + (y + z)$
3. $x + x = x$
4. $(x + y)z = xz + yz$
5. $(xy)z = x(yz)$

figure 2.1 Basic Process Algebra.

As in regular algebra · binds stronger than +. Furthermore we leave out brackets and the ·. Thus $(x \cdot y) + z$ becomes $xy + z$.

One might think that the axiom $x(y + z) = xy + xz$ is missing. However, this axiom was left out on purpose, because $x(y + z)$ represents something else then $xy + xz$, i.e. we want to consider models of the theory where they are different. The difference originates from the moment of choice. An example, due to P. Weijland, will explain this difference.

Suppose we are playing a game of Russian roulette. We start with putting just one bullet in the revolver's container and then swing the container. At this moment the *system*, in this case the revolver, 'knows' whether it will fire or not when the trigger is pulled. The outside world, however cannot tell the difference. The atomic actions involved in the sequel of this game are:

- trigger : the act of pulling the trigger of the revolver.
- bang : the sound of the bullet that gets fired.
- click : the sound of the revolver when the hammer hits an empty chamber.

Now there is a big difference between trigger · bang + trigger · click and trigger · (bang + click). The first expression models the actual situation. The outside world is only able to perform a trigger action and does not know what the result of this action will be. The second example models a situation in which we first pull the trigger and then let the system make a choice between bang and click , as if the container has to be swung again.

2.2 EXPLANATION OF SDF

In this section we will give a short introduction to SDF, the formalism we have used to define the syntax of PSF_d.

2.2.1 General Aspects

SDF [BHK89] stands for: 'Syntax Definition Formalism'. It is a language to specify the lexical syntax, context-free syntax and abstract syntax of programming languages in a formal way and can be seen as an alternative to LEX [LS79] and YACC [Joh79]. It is possible to generate a lexical scanner and some parse tables from such an SDF-definition [Rek87]. These parse tables together with a universal parser form a parser for the specified language. It is also possible to generate a so-called syntax directed editor from a description of the layout and the parse tables. This whole system is being implemented in LISP as part of ESPRIT Project 348: GIPE (Generation of Interactive Programming Environments).

2.2.2 SDF Syntax

An SDF definition consists of two parts: a *lexical syntax* and a *context-free syntax*. In both parts we deal with the notions *sort* and *function* that correspond, respectively, to non-terminals and to production rules as used in BNF grammars.

We will point out some of the SDF constructions that appear in the SDF specification of PSF_d. The *sorts* and *layout* declarations, in the lexical syntax section, introduce the lexical sorts while their *functions* declarations specify what kind of strings can be constructed over these sorts. Elements of the context-free syntax may be interspersed with strings belonging to the layout sorts. The latter will be skipped by the lexical analyzer generated from the SDF definition. The function declaration may be composed of other lexical sorts, (negated) character classes, terminals and list expressions. In the lexical syntax section two kinds of list expressions are allowed:

S*	zero or more occurrences of sort *S*
S+	one or more occurrences of sort *S*

In the function declaration of the context-free syntax section lexical sorts may be used as terminals of the grammar, though terminals may also be introduced directly, like "+" and "*" in the example. Moreover two more list expressions are allowed:

{S t}*	zero or more occurrences of sort *S*, separated by the terminal *t*.
{S t}+	one or more occurrences of sort *S*, separated by the terminal *t*.

The associativity of functions may be declared by means of the attributes: *assoc*, *left-assoc* and *right-assoc* while the attribute *par* can be added to the function declaration to state that the function may be surrounded by parentheses in order to change its priority.

2.3 EXPLANATION OF ASF

ASF is an algebraic specification formalism that emerged from the so-called 'PICO-formalism' [BHK85] and which is fully described in [BHK89]. An implementation of ASF is described in [Hend88].

We are using ASF as the basis for the modularization concepts of PSF_d and for the specification of abstract data types in PSF_d. Because most aspects of ASF specifications will appear in the description of PSF_d we will not discuss them here. For specific information we refer to [BHK89].

2.4 AN INFORMAL DESCRIPTION OF PSF_d

In this section we will give a description of all the features of PSF_d. These features are divided into three sections: modularization, specification of data types and specification of processes.

2.4.1 Modules

A PSF_d specification consists of a sequence of modules each one of which is either a *data module* or a *process module*. The data modules are used to define the properties of the data types and the process modules define the behaviour of the processes. Each module is given a unique name. PSF_d modules can be combined by *parameter binding* and *importing* only.

A module consists of a number of sections which are listed below.

DATA MODULE	*PROCESS MODULE*
parameter section	parameter section
export section	export section
import section	import section
sort section	atom section
function section	process section
variable section	set section
equation section	communication section
	variable section
	(process) definition section

figure 2.2 The different sections in modules.

In the next paragraphs we will explain the function of each section.

2.4.2 Lexical Syntax

In this paragraph we will describe the lexical syntax of PSF_d in an informal way.

- Layout:

  Possible layout characters are:
  - space
  - horizontal tabulation
  - carriage return
  - line feed
  - form feed

- Comments:

  Comments follow a layout character and begin with two hyphens and end with either an end of line (i.e. carriage return or line feed) or another pair of hyphens.

- Identifiers:

  Identifiers consist of a non-empty sequence of letters, digits or single quote characters, possibly with embedded hyphens.
  - examples : i, me, type-writer, prime', 'quotation', double--hyphen
  - non-examples: -x, -, x-

- Keywords:

  The following identifiers are reserved keywords:

atoms	end	merge	skip
begin	equations	module	sorts
bound	exports	of	sum
by	for	parameters	to
communications	functions	process	variables
data	hide	processes	when
definitions	imports	renamed	
encaps	in	sets	

  The names *hidden* and *export* are also forbidden as names for a *parameter* section.

- Operators:

  Operators are denoted by either a sequence of operator symbols or an identifier surrounded by dots. Possible operator symbols are: ! @ $ % ^ & + - * ; ? ~ / | \
  Some examples: &&, -?-, .push., %^@$

Layout characters and comments may separate identifiers in PSF_d but may never occur embedded in a lexical token. In cases of ambiguity, the longest token is preferred. For detailed information on the lexical syntax of ASF we refer to [BHK89].

2.4.3 Modularization

There are some constructs in PSF_d that help to make specifications in a modular fashion. The three sections that deal with this feature are the export, import and parameter section. Along with a short description of each modularization concept we will give the associated *structure diagrams*, as introduced in [BHK89].

Module *A* is represented by a rectangular box.

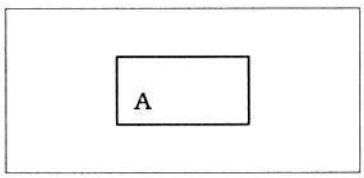

figure 2.3 Structure diagram of a module.

2.4.3.1 Export

All definitions that are listed in the export section are visible outside the module. A data module may define sorts and functions, while a process module may define atoms, processes and sets. All sorts, functions, atoms, processes and sets that are declared outside the export 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 the names in the export section of *B* are automatically exported by *A* too. This feature is called *inheritance*.

2.4.3.2 Import

The basic way to combine modules is by way of import. In the import section we define which modules have to be imported, possibly perform some renamings on the imported items and possibly bind parameters (see next section) of the imported module. By importing module *A* in module *B*, all exported *objects* in *A* become visible to *B*. The declaration of the importing module must be preceded by the declaration of the imported module in order to avoid cycles in the import graph. It is not allowed to import a process module in a data module.

Module *A* is imported by module *B*:

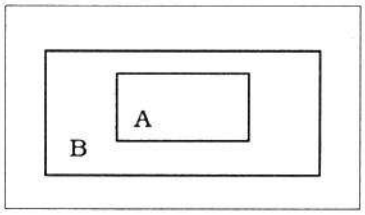

figure 2.4 Structure diagram of an import.

2.4.3.3 Parameters

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 lists some *formal* objects and each block has a name. Parameters in a data module may only consist of sorts and functions, whereas parameters in a process module may consist of atoms, processes and sets additionally. Whenever a parameterized module is imported into another module each parameter of the former module may become bound to a third module while all objects listed in the parameter are bound to actual sorts, functions, atoms, processes and sets from this third module. Not all parameters have to be bound when a module is imported. The unbound parameters are *inherited* by the importing module and are indistinguishable from the parameters defined in its own parameter section. Because parameter names cannot be renamed in PSF_d implicitly, all name clashes between a module's own and inherited parameters should be resolved by explicitly giving unique names to the parameters involved.

Parameters of a module are represented by ellipses carrying the name of the parameter. In the next example module *B* has a parameter *P*.

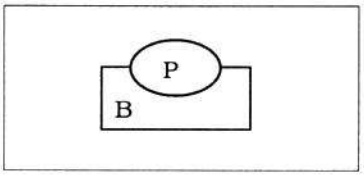

figure 2.5 Structure diagram of a module with a parameter.

2.4.4 Module Expressions

Module expressions are used inside the import section to rename visible names of the imported module and to bind formal to actual parameters.

2.4.4.1 Renaming

The visible names of a module can be renamed by 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. It is not possible to rename just one of the instances of an overloaded name. So if a renaming is applied to an overloaded name, all instances of this name will be renamed.

2.4.4.2 Parameter binding

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. Due to parameter binding a parameter is replaced by a name from the actual module as specified by the list of bindings. Therefore a parameter can only be bound once. Parameter binding should obey the following rules:

- The actual names must be visible outside the actual module.
- Formal names and actual names must be of the same kind, i.e., they both should be atoms, sorts etc. Furthermore their input type and output type should be the same.
- All names in a parameter that are not explicitly bound to a name of the actual module. are considered to be implicitly bound to the object in the actual nodule with the same name and type. All names of a parameter should be bound either explicitly or implicitly,

Binding of parameters is indicated by a line connecting the parameter and the actual module to which the parameter is bound. Parameter P of module B is bound to module A.

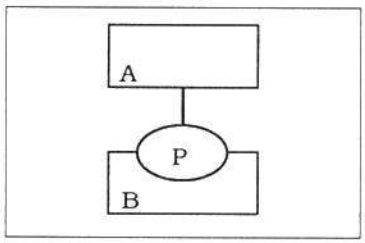

figure 2.6 Structure diagram of parameter binding.

2.4.5 Data Specification

There are some sections that are specific for the specification of data types. These sections are explained below. For more specific information about the specification of the data types we refer to [BHK89].

2.4.5.1 Sorts & Functions, Signatures

As we have seen, the declaration of sorts and functions can occur in two places. Declarations can occur in the export section of a module, so that they are visible, or they can be declared as being *hidden*. In the *sorts* section we define which sorts are introduced. The functions are declared within the *functions* section along with their *input type* (the type of the arguments) and their *output type*. The combination of an input type and an output type is simply called the *type* of a function. Functions without arguments will be called *constants*. Declarations of sorts and functions over these sorts are called *signatures*. See, for instance, [EM85] for a description of the notion of signatures.

2.4.5.2 Equations

To complete a data module we need a set of variables and a set of equations. The variables in a data module are typed with one of the sorts of the signature. With a set of typed variables and a signature it is possible to construct well-typed terms, i.e., terms that are constructed by type-wise correct composition of functions and variables.

An (unconditional) equation has the following form:

[*tag*] $t_l = t_r$

where t_l and t_r are well-typed terms of the same type.

Conditional equations can have two (equivalent) forms:

[*tag*] $t_{l_1} = t_{r_1}, \ldots, t_{l_n} = t_{r_n} \Longrightarrow t_l = t_r$ or

[*tag*] $t_l = t_r$ when $t_{l_1} = t_{r_1}, \ldots, t_{l_n} = t_{r_n}$

All equations occurring in a conditional equation must be made up of well-typed terms of the same type.

Variables in equations are implicitly universally quantified.

2.4.6 Process Specification

In this section we will describe the features of PSF_d that deal with the specification of processes. We will look at the definition of atomic actions, communication between atomic actions, processes and sets.

2.4.6.1 Sets

We introduce *sets* as a special feature in PSF_d in order to make specifications compact. A *set* is a collection of well-formed terms of the same sort, the sort associated to the set. Each set is given a unique name and is defined in the following way:

set-name = *set-expression*

There are several ways to construct a set-expression, which are listed below:

- Just the name of a *sort* denotes the set of all well-formed terms that are typed with the specified sort. Sorts do not have to be declared as sets.
- We are able to construct sets by enumerating terms: $\{ t_1, t_2, \ldots, t_n \}$
- Because it is impossible to enumerate infinite sets we need some weak form of *comprehension* in which the variables can only range over the domain of a given set.

So we introduce the so-called *placeholder* construction which is used in the next example to define a set A that consists of the terms that can be obtained by applying a certain function f to all elements of S: In this example t is the variable, which we will call a *placeholder* in the sequel.

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

In general the definition of a set by means of placeholders looks like:

$A = \{ t_1(\underline{u}), t_2(\underline{u}), \ldots, t_n(\underline{u}) \mid u_1 \text{ in } D_1, u_2 \text{ in } D_2, \ldots, u_m \text{ in } D_m \}$

where $t_i(\underline{u})$ means that all free variables of t_i are among $u_1, u_2, \ldots, u_m$, D_i may be either a sort or the name of a set, already declared in the *sets* section, and u_i acts as a placeholder for an arbitrary term or element of D_i.

The enumeration construction, as introduced above, can be looked upon as a special form of the placeholder construction in which the terms t_i do not contain any free variables. In this case we do not use any placeholders so the vertical bar disappears.

- Finally there are three binary operators on sets of the same type:
  - Union: $s_1 + s_2$
  - Intersection: $s_1 . s_2$
  - Difference: $s_1 \setminus s_2$

  The + and . operator are associative and the \ operator is left-associative. All operators have the same precedence, however precedence can be forced using parentheses.

2.4.6.2 Atomic Actions

The atomic actions that are used to describe a process are listed in the *atoms* section. The atomic actions resemble the functions from the data section in some respects. They possibly have some arguments but they do not have an output type, as functions do.

An example of the declaration of some atomic actions where a_i stands for the name of an atomic action and s_i stands for a sort:

$a_1, a_2 \quad : s_1$
a_3
$a_4 \quad : s_2 \# s_3$

To be more specific, we will not call a construction an atomic action until all arguments are substituted by a term. So $a_4 : s_2 \# s_3$ is in fact the mere definition of a scheme to generate atomic actions rather than an atomic action itself.

There is one implicitly defined sort called: *atoms*. This sort is only available in the process specification part and can be used in constructing sets of atoms like in the next example:

```
sets
  of atoms
    H = { send(n), read(n) | n In NATURAL }
```

figure 2.7 Example of the use of the predefined sort *atoms*.

2.4.6.3 Communication

Communication in PSF_d can occur between atomic actions only. In such a communication exactly three atomic actions are involved. Two atomic actions which communicate and one that is the resulting *communication action*, i.e. we have *handshaking* communication. In the *communications* section we define which atoms can communicate and what the result of this communication will be.

An example of such a *communication definition* in which a,b,c stand for atomic actions:

$$a \mid b = c$$

Communication is commutative so the definition of $a \mid b = c$ implies $b \mid a = c$. In the next example we want to express the fact that the communication of an a action and a b action which both operate on elements of the set S results in a c action. We will use the placeholder again:

$$a(d) \mid b(d) = c(d) \text{ for } d \text{ in } S$$

Beware of the difference between this example, where each $a(d)$ action communicates with one specific $b(d)$ action and where d stands for the same term in both actions, and the next example:

$$a(d) \mid b(e) = c(d) \text{ for } d \text{ in } S, e \text{ in } S$$

A communication definition should be given for all atomic actions that are visible within a module. Whenever a communication is not listed in the *communications* section, it is thought of as being a communication resulting in deadlock (see [BK86c]).

2.4.6.4 Variables

The variables in a process module can range over a sort or a set. The scope of a variable is the whole *definitions* section, unless a variable is temporarily overridden due to the use of a placeholder with the same name. Each variable in the *variables* section should have a unique name.

2.4.6.5 Processes

Processes have to be declared in the *processes* section along with the type of their possible arguments.

An example in which P_i stands for a process name and s_i stands for a sort:

$$P_1$$
$$P_2 \quad : \quad s_1 \# s_2$$

In the *definitions* section the behaviour of the processes, that have been declared in the *processes* section first, is defined. An example of such a definition in which P stands for a process name, a_i stands for an argument and PE stands for a *process-expression*:

$$P(a_1,a_2, \ldots, a_n) = PE$$

Each argument is a term of the right type, possibly containing variables which are defined in the *variables* section. The process name with a possible list of arguments is called the *process definition head*.

Process expressions are defined by means of induction:

- Each atomic action is a process-expression.
- There is one predefined process-expression called *skip* that represents the pre-abstraction from ACP. This feature was introduced in [BB87] where the atomic action t is used.
- There are three binary operators on process-expressions:
  - sequential composition : $PE_1 \,.\, PE_2$
  - alternative composition or choice: $PE_1 + PE_2$
  - parallel composition or merge : $PE_1 \,||\, PE_2$

  These operators are all associative. Sequential composition has precedence over parallel composition which in turn has precedence over alternative composition.
- There are two constructions that use the placeholder:
  - summation : sum(v in S, $PE(v)$)
    which generalizes alternative composition.
    For a finite set S, where $S = \{v_1, \ldots, v_n\}$, this is an abbreviation of:
    $PE(v_1) + PE(v_2) + \ldots + PE(v_n)$
  - merge : merge(v in S, $PE(v)$)
    which generalizes parallel composition.
    For a finite set S, where $S = \{v_1, \ldots, v_n\}$, this is an abbreviation of:
    $PE(v_1) \,||\, PE(v_2) \,||\, \ldots \,||\, PE(v_n)$
- Finally, there are two constructions that operate on a set of atoms and a process-expression:
  - encapsulation : encaps(S, PE)
  - pre-abstraction : hide(S, PE)

  Note that we will not need the ACP constant δ or the (auxiliary) operators $\|$, $\mathbb{L}$ (as in [BB88]).

2.4.6.6 Scope of placeholders

We have pointed out the use of placeholders in some of the constructions we mentioned earlier. Still we have not defined the *scope* of the placeholder. In the following examples the placeholder along with its scope are underlined.

- Sets : The scope is limited to the enclosing braces.
  $A = \{ f(\underline{t}) \mid \underline{t} \text{ in } S \}$
- Communication : The scope is limited to the communication definition preceding the placeholder definition.
  $a_1(\underline{d}) \mid a_2(\underline{d}) = b(\underline{d})$ for $\underline{d}$ in S
- Processes : The scope is limited to the enclosing parentheses and can be overridden by a placeholder definition on a lower level.
  $X = x + y \cdot \text{sum}(\underline{d} \text{ in } S, \underline{r(d) \cdot z + \ldots} \text{ merge}(\underline{d} \text{ in } D, \underline{Y(d)}) \ldots)$

2.5 DEFINITION OF PSF_d IN SDF

2.5.1 Why use SDF?

In this section we will give the definition of the PSF_d formalism in SDF. The reason that we have chosen to use SDF instead of, for instance, a BNF grammar is that we found the former to be much more formal and that we had the possibility to check the PSF_d grammar during the development by means of the error messages generated by the parser generator described in [Rek87]. Moreover, we could easily build a prototype parser that was able to check our PSF_d specifications against any grammatical inconsistencies.

2.5.2 Definition of PSF_d

Beware that this definition of PSF_d does not contain the syntax of the data modules. We refer to [BHK89] for a description of the syntax of the algebraic specifications in ASF. Nevertheless, the syntax for the modularization concepts, which is borrowed from ASF, is included.

```
module PSF
begin

  lexical syntax

    sorts
      id-char, id-body, ident,
      op-symbol, operator,
      com-char, com-end

    layout
      white-space, comment

    functions

      [0-9a-zA-Z']                              -> id-char
      [0-9a-zA-Z'\-]                            -> id-body
      id-char                                   -> ident
      id-char id-body* id-char                  -> ident

      [!@$%\^&+\-*;?~/|\\]                      -> op-symbol
      op-symbol+                                -> operator
      "." ident "."                             -> operator

      [ \n\t\f\r]                               -> white-space

      ~ [\n\-]                                  -> com-char
      "-" ~ [\n\-]                              -> com-char
      "--"                                      -> com-end
      "-\n"                                     -> com-end
      "\n"                                      -> com-end
      "--" com-char* com-end                    -> comment

  context-free syntax

    sorts
      specification, process-module, parameters, parameter, exports,
      imports, module-expression, modifier, renamed,
      renamings, renaming, binding, bound, psf-sorts,
      psf-functions, psf-function, atoms, atom-declaration, processes,
      process-decl, sets, set-defs, set-definition,
      set-exp, placeholder,
      set-item, variables, vars,
      communications, communication, communication-def,
      atom-exp, definitions,
```

```
  process-def, process-def-head,
  process-exp, predef-process, set-operator,
  identifier, ident-or-op,
  term, primary

functions
   ident                                              -> identifier
   process-module+                                    -> specification
   "process" "module"  identifier
      "begin"
         parameters
         exports
         imports
         atoms
         processes
         sets
         communications
         variables
         definitions
      "end" identifier                                -> process-module

   "parameters" { parameter "," }+                    -> parameters
                                                      -> parameters
   identifier
   "begin"
      psf-sorts
      psf-functions
      atoms
      processes
      sets
   "end" identifier                                   -> parameter

   "exports"
      "begin"
         atoms
         processes
         sets
      "end"                                           -> exports
                                                      -> exports

                                                      -> imports
   "imports" {module-expression ","}+                 -> imports
   identifier                                         -> module-expression
   identifier "{" modifier "}"                        -> module-expression
   renamed                                            -> modifier
   bound                                              -> modifier
   renamed bound                                      -> modifier
   bound renamed                                      -> modifier

   "renamed" "by" renamings                           -> renamed
   "[" {renaming ","}+ "]"                            -> renamings
   ident-or-op "->" ident-or-op
                                                      -> renaming
   identifier                                         -> ident-or-op
   "_" operator "_"                                   -> ident-or-op
   operator "_"                                       -> ident-or-op

   binding+                                           -> bound
   identifier "bound" "by" renamings "to" identifier
                                                      -> binding

   "sorts" {identifier ","}+                          -> psf-sorts
                                                      -> psf-sorts

   "functions" psf-function+                          -> psf-functions
                                                      -> psf-functions
```

```
identifier ":" {identifier "#"}* "->" {identifier "#"}+
                                                -> psf-function
operator "_" ":" identifier "->" {identifier "#"}+
                                                -> psf-function
"_" operator "_" ":" identifier "#" identifier
                      "->" {identifier "#"}+    -> psf-function


"atoms" atom-declaration+                       -> atoms
                                                -> atoms
{identifier ","}+                               -> atom-declaration
{identifier ","}+ ":" {identifier "#"}+         -> atom-declaration

"processes" process-decl+                       -> processes
                                                -> processes
{identifier ","}+                               -> process-decl
{identifier ","}+ ":" {identifier "#"}+         -> process-decl

"sets" set-defs+                                -> sets
                                                -> sets

"of" identifier set-definition+                 -> set-defs
"of" "atoms" set-definition+                    -> set-defs
identifier "=" set-exp                          -> set-definition

identifier                                      -> set-exp
"{" { set-item "," }+ "}"                       -> set-exp
set-exp "+" set-exp                             -> set-exp
                                                        {par,assoc}
set-exp "." set-exp                             -> set-exp
                                                        {par,assoc}
set-exp "\" set-exp                             -> set-exp {par}
"{" {set-item ","}+ "|" {placeholder ","}+ "}"  -> set-exp

identifier                                      -> set-item
identifier "(" { term "," }+ ")"                -> set-item

identifier "in" identifier                      -> placeholder

"variables" vars+                               -> variables
                                                -> variables
{ identifier "," }+ ":" "->" identifier         -> vars

"communications" communication-def+             -> communications
                                                -> communications
communication                                   -> communication-def
communication "for" {placeholder ","}+          -> communication-def
atom-exp "|" atom-exp "=" atom-exp              -> communication

identifier                                      -> atom-exp
identifier "(" { term "," }+ ")"                -> atom-exp

"definitions" process-def+                      -> definitions
                                                -> definitions
process-def-head "=" process-exp                -> process-def
identifier                                      -> process-def-head
identifier "("  { term "," }+ ")"               -> process-def-head

predef-process                                  -> process-exp
process-def-head                                -> process-exp
process-exp "." process-exp                     -> process-exp
                                                        {par,assoc}
process-exp "+" process-exp                     -> process-exp
                                                        {par,assoc}
process-exp "||" process-exp                    -> process-exp
                                                        {par,assoc}
```

```
        primary                                                -> term
        term operator primary                                  -> term
        identifier                                             -> primary
        identifier "(" {term ","}+ ")"                         -> primary
        operator primary                                       -> primary
        "(" term ")"                                           -> primary

        "sum" "(" placeholder "," process-exp ")"              -> process-exp
        "merge" "(" placeholder "," process-exp ")"            -> process-exp
        set-operator "(" identifier "," process-exp ")"        -> process-exp

        "skip"                                                 -> predef-process
        "encaps"                                               -> set-operator
        "hide"                                                 -> set-operator

end PSF
```

figure 2.8 Specification of PSF_d.

2.6 SEMANTICAL CONSTRAINTS

There are some constraints imposed on PSF_d specifications that we are not able to express in SDF. This concerns *overloading*, restriction on communication and binding of variables.

2.6.1 Overloading

It is allowed to *overload* the names of functions, atoms and processes. This means that the same name can be used to denote different functions, atoms or processes.

An example for a function *f*:

f: s1 # s2 -> s3
f: s3 -> s3

A similar example can be constructed for atoms and processes, though the latter do not have an output-type. When the function name *f* occurs in a certain term we will have to determine which function *f* was meant by looking at the types of the arguments *f* is applied to. It is possible to disambiguate each overloaded name by postfixing it with its input-type, i.e. its arguments. The names used in the sequel for functions, atoms and processes will be *disambiguated* names.

Overloaded functions, atoms and processes should have unique input types. This restriction forbids overloaded constants and multiple declaration of names with the same type. Variables and sets cannot be overloaded. This restriction forbids multiple declaration of variable names and set names within one module.

2.6.2 Typing of terms

Type assignment of a term is performed by inside-out typing. This can be achieved by first determining the type of the constants and variables in a term and thereafter propagating this type information outward to the enclosing terms until the type of the complete term has been determined. The uniqueness of types in each stage of this process is guaranteed by the restriction that the sets of names of as well constants and variables as sorts and sets must be disjoint, and by the restrictions placed on overloaded functions and variables.

2.6.3 Communication

There are three restrictions imposed on the definition of communications. The first is *firm handshaking*, the second considers the consistency of export of atoms involved in communication actions and the third deals with the consistency of communications when combining modules.

2.6.3.1 Firm Handshaking

All communications must satisfy *handshaking*. This means that no atomic action that is the result of a communication is able to communicate itself with some other atomic action. To be able to check this property we demand that an atomic action $a(\underline{v})$, with a possibly empty list of arguments v_i may not occur on the left as well as the right hand side of the equation sign in a list of communication definitions. We call this *firm handshaking* because it is more restrictive than *handshaking*. It forbids, for instance, the following definition:

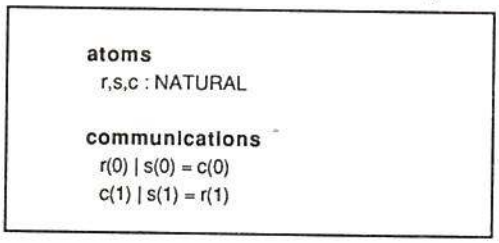

figure 2.9 A violation of firm handshaking.

However, we think of this as bad programming style anyway.

2.6.3.2 Consistency of Export

The second restriction on communications deals with visibility, outside a module, of the atoms involved in a communication. Whenever two atoms that are able to communicate with each other are exported from a module, the atom that is the result of this communication must be exported too. This restriction forbids the situation in which it is possible to have a communication between two (visible) atomic actions, but subsequently not being able to *see* the result of this communication.

2.6.3.3 Consistency of Communications

The definition, in separate modules, of the result of a communication may lead to inconsistencies when putting these modules together. We call two modules inconsistent with respect to their communications whenever there exists a communication between two atomic actions that is defined in both modules and yields two different atomic actions. In figure 2.10 two examples of such an inconsistency are shown:

In the first example modules A and B are inconsistent with respect to their communications, because A defines $a \mid b$ to be c and B defines $a \mid b$ to be d. In determining whether two modules are consistent we should not only consider the explicitly defined communications, but also the assumption that all communications, between atomic actions that are visible in a module, that have not been defined in the *communications* section, are implicitly defined to be deadlock. The second example in fig. 2.10 illustrates this situation.

Module A does not list a communication between a and d, and so this communication is defined to yield deadlock. However module B tries to re-define this communication by $a \mid d = e$, which is illegal.

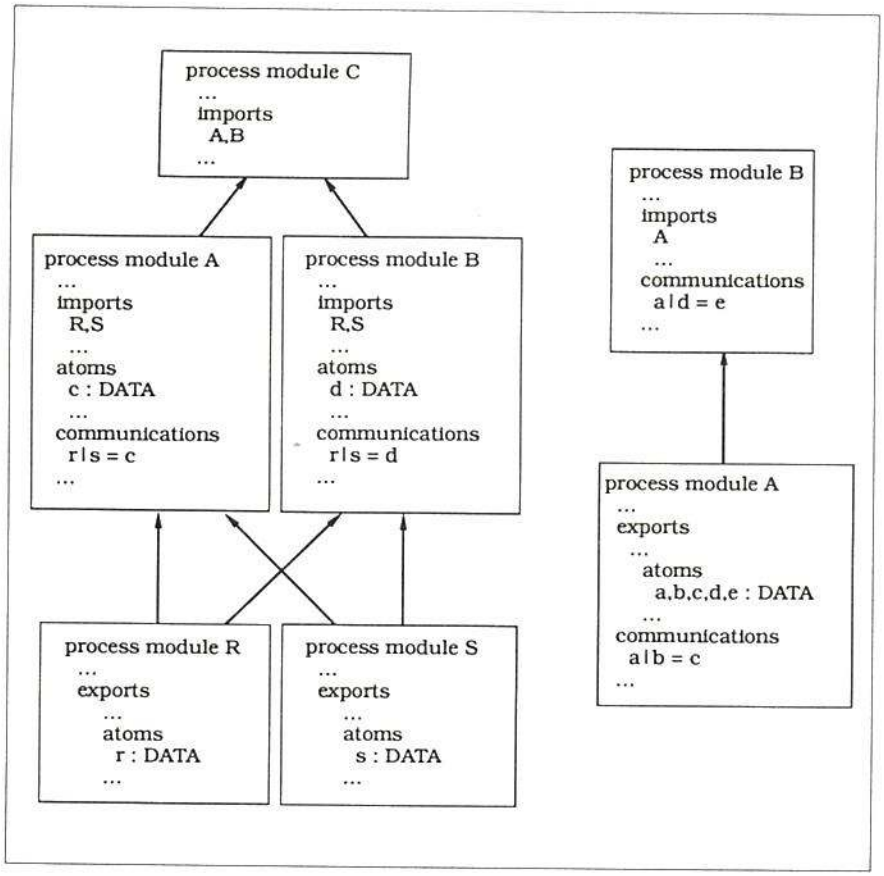

figure 2.10 Two possible sources of inconsistency.

It is not allowed to combine two inconsistent modules. To be more precise, the following situations are not allowed:

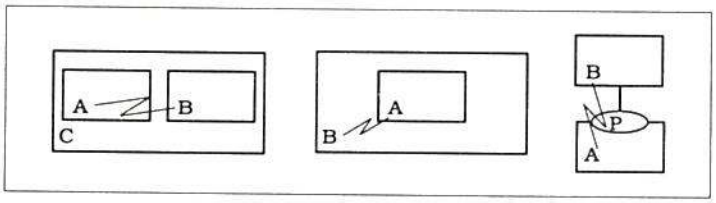

figure 2.11 Three ways of illegally combining inconsistent modules.

Whenever two modules are inconsistent they may not be
- imported into a third module.
- imported into each other.
- bound to each other's parameters.

2.6.4 Variables

All variables that occur in a process-expression, i.e. on the right hand side of the equation sign in a process definition, should be bound. This binding can be obtained in one of two ways:

- the variable belongs to a placeholder construction in which this variable was introduced.

- the variable already occurred in one of the arguments of the process definition head, i.e. on the left hand side of the equation sign.

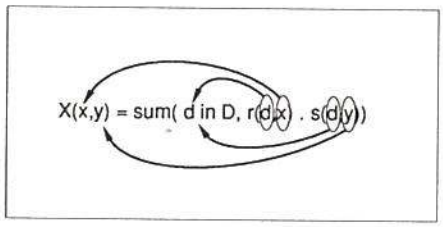

figure 2.12 Binding of variables in a process expression.

3 SEMANTICS

In this section we define the semantics of PSF_d, thus giving a meaning to PSF_d specifications.

3.1 SEMANTICS FOR PSF_d

Due to the nature of PSF_d, being a mixture of two different formalisms, it is not possible to assign one uniform semantics to the language and so its definition will break up into four sections.

First, we have to define the semantics for the data specification part. It is quite natural to choose the same semantics as the one chosen for ASF, i.e. the *initial algebra semantics* as in [EM85] and [GM85]. Ideally, the set of equations takes the form of a complete term rewriting system, so that equality of terms can be determined by reducing to *normal form*, and the set of normal forms is isomorphic to the initial algebra. For the section that defines the processes it is convenient to use another kind of semantics because, by nature, it has more in common with *transition networks*. (Note, on the other hand, that in [Mau87a] we find an attempt to give an algebraic specification of ACP using initial algebra semantics.) For this process section we use an *operational semantics* that is defined with the aid of *action relations* [Plo82], which will be presented in section 3.7.3. Action relations have already been introduced, for ACP, in [Gla87] and have been used in [BB88] to introduce ACP with action relations instead of *axioms*. On top of these action relations we can define a semantics, like e.g. *bisimulation semantics* or *failure semantics*. Finally we also give a semantics for the sets and the atomic actions, which

will both be derived from the initial algebra semantics. The dependencies among the semantics of the different parts of PSF_d are expressed by the following picture:

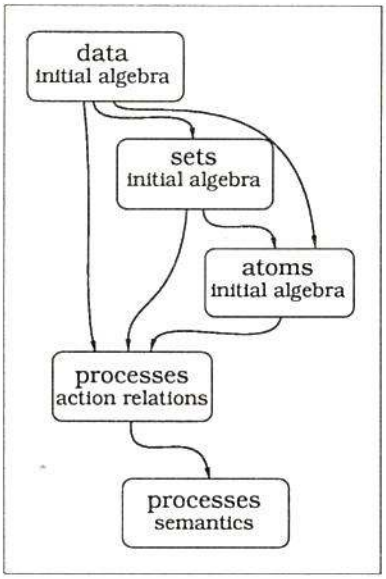

figure 3.1 Dependencies among different semantical domains.

We will discuss the semantics in order of dependency starting with the semantics for data types. However, we have to make sure that modules are in some kind of normal form before we can treat the semantics, so we treat the origin rule and normalization in the next two paragraphs.

3.2 THE ORIGIN RULE

Because a PSF_d specification may consist of several modules, there may arise some problems with multiple declarations of the same name when putting these modules together in case of import. We don't want any unintended and unexpected name identifications, so we introduce the so-called *origin rule*, in a similar way as is done in ASF [BHK89] to locate the defining position of each occurrence of an identifier.

To each name a of an identifier we encounter in a module we assign an origin in the form of a tuple $<t,m,s,c,n>$ which gives information about the textual position where a certain name n, to which a 'owes its existence', has been declared. The parameters in the tuple stand for:

- •t: The type of the module in which the declaration of n occurs:
  t = *data* for a data module
  t = *process* for a process module
- •m: The name of the module in which the declaration of n occurs;
- •s: The section of the module in which the declaration of n occurs:

s = <*p-name*> for a parameter section with name *p-name*,
s = *export* for the export section,
s = *hidden* for the sort, function, atom, process, set and variable sections outside the export section.

•c: The subcategory to which n belongs:
c = *sort* for a sort name,
c = *function* for a function name,
c = *atom* for an atom name,
c = *process* for a process name,
c = *set* for a set name,
c = *variable* for a a variable name.

•n: The name as introduced by the declaration.

The origin of a certain name propagates in the following way:

- *Declaration:* When a name a is declared, it obtains origin <t,m,s,c,n>, where t, m, s and c are determined from the context of the declaration and initially $n = a$.
- *Import:* Import of a name does not affect its origin.
- *Renaming:* A name introduced by a renaming inherits the origin of the name it replaces.
- *Parameter binding:* The origin of an actual name does not change by binding a formal name to it. The origin of the formal name disappears along with the formal name itself.

The origin rule:

- Two visible sorts, functions, atoms, processes or sets are identical if they both have the same name and the same origin. Visible sorts, functions, atoms, processes and sets having the same name but different origin are forbidden.
- Two hidden sorts or hidden sets are identical if they have the same origin.
- Two variables are identical if they have the same origin and if the corresponding types (sorts) can be identified using the aforementioned rules.
- Two hidden functions, atoms and processes are identical if they have the same origin and if the two corresponding types have equal structure and can be identified componentwise using the first two rules given above.

Due to the origin rule multiple import of the same module, via different routes, is allowed, but clashes of identical (disambiguated) names originating from different modules are forbidden. When two modules are combined, the hidden names of the modules are *implicitly* renamed to avoid name classes.

3.3 NORMALIZATION

In order to be able to assign a semantics to a PSF_d specification we have to assign a semantics to each module. The semantics of a module can only be determined in its context, being the total

specification. This evaluation of a module in its context leaves us with a so-called *normal form*.

In evaluating a module, as many *imports* and *parameter bindings* as possible are eliminated. Because each PSF_d specification consists of two types of modules, it is quite natural to extend this division into the notion of the normal form of a module. This means that after evaluating a module each normal form consists of one process module and one data module which is imported in the former.

How this normalization should be performed, is still an open problem. However, we think it should follow the normalization procedure for data modules, as given in [BHK89].

3.4 SEMANTICS FOR DATA TYPES

As the semantics for the data types we use the initial algebra semantics as defined in [EM85, GM85]. We assume that all modularization concepts from the data modules have been removed by the normalization procedure, thus leaving a *flat* algebraic specification. To define this initial algebra we first need to introduce some other notions.

- A *signature* Σ is a collection of names of sorts and functions. To each function name we associate a list of sort names that represent the input type and one sort name for the output type. Functions with an empty input type are constants of the specified output type.
- The set V consists of variables. To each variable a sort is associated.
- A *term* is a construction of functions and variables with correctly typed arguments, defined inductively:
  - Each variable associated with sort S, is a term of sort S.
  - If $t_1 \ldots t_n$ are terms of sort $S_1 \ldots S_n$, and function f has input type $S_1 \times \ldots \times S_n$ and output type T, then $f(t_1, \ldots, t_n)$ is a term of sort T.

  A term containing no variables is called a *closed term*, as opposed to an *open term*.
- An *equation* is a pair of two terms of the same sort. For example: $t_1 = t_2$. Variables in equations are universally quantified.
- An *equational specification* (Σ,E) consists of a signature Σ and a set of equations E.
- *Derivability*, of an equality of two terms of an algebraic specification, $(\Sigma,E) \vdash t_1 = t_2$, is inductively defined by:
  - $(\Sigma,E) \vdash t_1 = t_2$ if $t_1 = t_2 \in E$.
  - $(\Sigma,E) \vdash t = t$.
  - $(\Sigma,E) \vdash t_1 = t_2$ if $(\Sigma,E) \vdash t_2 = t_1$.
  - $(\Sigma,E) \vdash t_1 = t_3$ if $(\Sigma,E) \vdash t_1 = t_2$ and $(\Sigma,E) \vdash t_2 = t_3$.
  - $(\Sigma,E) \vdash \theta(t_1) = \theta(t_2)$ if $(\Sigma,E) \vdash t_1 = t_2$, with θ a substitution of variables.
  - $(\Sigma,E) \vdash C[t_1] = C[t_2]$ if $(\Sigma,E) \vdash t_1 = t_2$, with C[...] a context.
- A *Σ-algebra* is a structure with an *interpretation* of every sort and function from Σ. The interpretation of a sort is a set and an interpretation of a function is a correctly typed function defined on these sets. The interpretation of a closed term is defined using the following:

  $[\![f(t_1,\ldots,t_n)]\!] = [\![f]\!]([\![t_1]\!],\ldots,[\![t_n]\!])$

- An equation of two closed terms is true in a Σ-algebra A, whenever the interpretation of both terms denotes the same element.

  $A \models t_1 = t_2 \quad \Leftrightarrow [t_1] =_A [t_2]$

  When all equations, $t_1 = t_2$, in E, are valid in the Σ-algebra A, we write $A \models E$.

- The class of Σ-algebras A with $A \models E$. is denoted by $Alg(\Sigma,E)$. This class contains one special algebra called the *initial algebra* of (Σ,E), $I(\Sigma,E)$.

  The initial algebra is the algebra that satisfies two requirements, namely:

  - *No junk.* This means that each element in the Σ-algebra is the interpretation of some term over the signature, so there are no unnamed elements of the Σ-algebra.
  - *No confusion.* This means that equations of closed terms in $I(\Sigma,E)$ are only valid when they can be derived from the specification E.

  $I(\Sigma,E) \models t_1 = t_2 \quad \Rightarrow (\Sigma,E) \vdash t_1 = t_2$

3.5 SET SEMANTICS

Because we have defined a very simple notion of sets, it is both intuitively and formally simple to give a meaning to it. The initial algebra generated by the sort associated with the set is considered to be the basis. Sets are given a meaning by interpreting them as parts of the initial algebra. Every sort by itself defines the set of all elements in its initial algebra. The set constructed by enumerating some terms over the signature of a sort S is just the set of equivalence classes of these terms. In the same way one can define the union, intersection and difference operators by applying these operations to the sets of corresponding elements in the initial algebra.

Formally:

Let S be a sort, and let for every term t over the signature of S, $[t]_S$ be defined as the corresponding element in the initial algebra of S. Thus $[t]_S$ (or for short $[t]$) is the equivalence class of all terms equal to t. We assume that for each element a in the initial algebra we can find a representative t (such that $[t] = a$). For each set D in the initial algebra we will denote a set of representatives of all elements in D by $Repr(D)$.

Then we define the interpretation of a set of sort S in the initial algebra (IA) inductively by:

- $[S] = \text{IA}$;
- $[\{t_1, \ldots, t_n\}] = \{[t_1], \ldots, [t_n]\}$ for terms $t_1, \ldots, t_n$ over the signature of S;
- $[s_1 + s_2] = [s_1] \cup [s_2]$;

  $[s_1 . s_2] = [s_1] \cap [s_2]$;

  $[s_1 \setminus s_2] = [s_1] \setminus [s_2]$ for sets s_1 and s_2 of sort S.
- $[\{t_1(\underline{u}), \ldots, t_n(\underline{u}) \mid u_1 \in D_1, \ldots, u_m \in D_m\}] = [\{t_1(\underline{u}) \mid u_1 \in D_1, \ldots, u_m \in D_m\}] \cup \ldots \cup [\{t_n(\underline{u}) \mid u_1 \in D_1, \ldots, u_m \in D_m\}]$
- $[\{t(\underline{u}) \mid u_1 \in D_1, \ldots, u_m \in D_m\}] = \{[t(\underline{u})] \mid u_1 \in \text{Repr}(D_1), \ldots, u_m \in \text{Repr}(D_m)\}$

3.6 SEMANTICS FOR ATOMIC ACTIONS

The atomic actions resemble the functions from the data modules, though atomic actions do not have an output type. Because of this similarity we want to define the semantics of the atomic actions in the same way as the functions, namely by means of the initial algebra semantics. We define an equivalence relation on atomic actions in the following way:

$a(v_1, v_2, \ldots, v_n) = b(u_1, u_2, \ldots, u_m)$ whenever

- the name a is equal to the name b
- $m=n$
- the input types of a and b are equal
- $\forall i, 1 \leq i \leq n$: $v_i = u_i$, in the initial algebra of the sort of v_i and u_i .

This construction corresponds with the initial algebra obtained by extending the algebraic specification of the data types with a new sort *atom* and adding for each name of an atom, a corresponding function.

3.7 OPERATIONAL SEMANTICS

3.7.1 Action Rules

In this section we will define the operational semantics for the process definition part of PSF_d with the aid of so-called *action rules*. These action rules have been used already in other concurrency theories, see for example [Plo82] in which Plotkin gives the operational semantics for CSP [Hoa85]. Action rules in ACP are introduced in [Gla87]. But first we will have a look at what a process definition stands for.

3.7.2 Process Definitions

A process definition in general looks like:

- $X(t_1(\underline{v}), \ldots, t_n(\underline{v})) = y(\underline{v})$;

  $\underline{v}$ is a list of variables declared in the *variables* section.
  t_i is a term from the data specification part, possibly containing some variables from the list $\underline{v}$.
  X is a process name from the process definition part.
  y is a process expression.

All closed data terms occurring in a process definition should be looked upon as a notation for the corresponding equivalence class of this term, in the initial algebra. It would have been more accurate if we would have written a term t as $[t]$. However, we leave out the brackets for reasons of simplicity.

There are no differences between the process expressions in figure 3.2. These are just different ways of writing: send([0]).X([0]):

- send(0).X(0)
- send(0+0).X(0)
- send(0).X(0+0)

figure 3.2 Three different representations of the same process expression.

All process definitions that contain variables, which must be bound properly, are an abbreviation of a possibly infinite series of process definitions in which all variables have been eliminated. This series is constructed by replacing all occurrences of a certain variable v, of sort S, with a representative of each equivalence class of the initial algebra of S.

The next example will clarify this notion. Suppose we have the following fragment of a specification :

```
processes
  X, Y: BOOLEAN # NATURAL

variables
  b : -> BOOLEAN
  n : -> NATURAL

definitions
  X(b,n) = send(n) . Y(b,n)
```

figure 3.3 An abbreviation of process definitions.

Then the *definitions* section represents, a.o.:

```
X([true],[0]) = send([0]) . Y([true],[0])
X([true],[s(0)]) = send([s(0)]) . Y([true],[s(0)])
X([true],[s(s(0))]) = send([s(s(0))]) . Y([true],[s(s(0))])
...
X([false],[0]) = send([0]) . Y([false],[0])
X([false],[s(0)]) = send([s(0)]) . Y([false],[s(0)])
...
```

figure 3.4 Part of the expanded *definitions* section.

Though we are writing process definitions as equations like $X = a$, $X = b$, we merely mean that X has a summand a and a summand b. So whenever the same left-hand side of an equation occurs more than once, possibly due to expanding the *definitions* sections as described above, we consider the corresponding right-hand sides as alternatives. In this way we can make a non-deterministic choice between the alternative right-hand sides, just like applying the +-operator.

Note that an alternative decision could have been to forbid this situation and to rename such an expression into deadlock, indicating an error. It is possible that a future implementation of a simulator would support both modes of execution and would let the user choose between them.

Whenever a process name with closed terms for all its arguments does not occur in a expanded specification as a left-hand side, it is considered to be equal to deadlock. We recall that δ is the neutral element for alternative composition.

3.7.3 Action Rules for PSF_d

In the following section we present the action rules for PSF_d.

For each element $[a]$ of the initial algebra of atomic actions we define a binary relation $\xrightarrow{[a]}$ and a unary relation $\xrightarrow{[a]}\surd$ on closed process expressions. If a is an atomic action, and $[a]$ its equivalence class (so $[a] \in$ IA), we write $\xrightarrow{a}$ instead of $\xrightarrow{[a]}$.

$x \xrightarrow{a} y$ means that the process expression represented by x can evolve into y, by executing the atomic action $[a]$.

$x \xrightarrow{a} \surd$ means that the process expression represented by x can terminate successfully after having executed the atomic action $[a]$. The special symbol $\surd$ can be looked upon as a symbol indicating successful termination of a process.

The relations $\xrightarrow{a}$ are generated by the rules in the following tables, i.e. $x \xrightarrow{a} y$ only holds if this can be derived using these rules.

In the following tables we will use some symbols that have a special meaning. These symbols are:

- a,b,c : atomic actions or *skip*.
- x,y,x',y' : variables on processes, i.e. we can substitute any process for these variables.

Along with some of the rules we will give an explanation:

- *R.||*
  - $a \mid b = c$ means that the communication between a and b has been defined to be c.
- *R.encaps*:
  - H : the set of atomic actions that have to be encapsulated.
  - $a \notin H$ means $\forall x \in H : x \neq a$. There is no atomic action in H which is, according to the initial algebra semantics, equal to a.
- *R.hide*:
  - I : the set of atomic actions that have to be renamed into *skip*.
  - $a \in I$ means $\exists x \in I : x = a$. There is an atomic action in I which is, according to the initial algebra semantics, equal to a.
  - $a \notin I$ means $\forall x \in I : x \neq a$. There is no atomic action in I which is, according to the initial algebra semantics, equal to a.
- *R.rec*:
  - $\underline{u} \in \underline{D}$ means $u_1 \in D_1, u_2 \in D_2, \ldots, u_n \in D_n$
    - $\underline{u} = (u_1, u_2, \ldots, u_n)$
    - $\underline{D} = (D_1, D_2, \ldots, D_n)$
    - D_i is a sort.
  - $y(\underline{u})$: a process expression with a list of terms $\underline{u} \in \underline{D}$ as parameters.
  - X : a process name declared in the *processes* section as $X : D_1 \# D_2 \# \ldots \# D_n$

- $X(\underline{u}) = y(\underline{u})$: an equation from the *definitions* section.

- *R.sum*:
  - D : a set.
  - $u \in D$: u is some specific element of D.
  - d in D: d is a variable of set D.

- *R.merge*:
  - $|D|$: the number of elements in set D.

R.a	$a \xrightarrow{a} \surd$		
R.+1	$\dfrac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'}$	R.+2	$\dfrac{x \xrightarrow{a} \surd}{x+y \xrightarrow{a} \surd}$
R.+3	$\dfrac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'}$	R.+4	$\dfrac{y \xrightarrow{a} \surd}{x+y \xrightarrow{a} \surd}$
R.·1	$\dfrac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y}$	R.·2	$\dfrac{x \xrightarrow{a} \surd}{x \cdot y \xrightarrow{a} y}$
R.||1	$\dfrac{x \xrightarrow{a} x'}{x \Vert y \xrightarrow{a} x' \Vert y}$	R.||2	$\dfrac{x \xrightarrow{a} \surd}{x \Vert y \xrightarrow{a} y}$
R.||3	$\dfrac{y \xrightarrow{a} y'}{x \Vert y \xrightarrow{a} x \Vert y'}$	R.||4	$\dfrac{y \xrightarrow{a} \surd}{x \Vert y \xrightarrow{a} x}$
R.||5	$\dfrac{x \xrightarrow{a} x';\ y \xrightarrow{b} y';\ a\|b=c}{x \Vert y \xrightarrow{c} x' \Vert y'}$	R.||6	$\dfrac{x \xrightarrow{a} \surd;\ y \xrightarrow{b} y';\ a\|b=c}{x \Vert y \xrightarrow{c} y'}$
R.||7	$\dfrac{x \xrightarrow{a} x';\ y \xrightarrow{b} \surd;\ a\|b=c}{x \Vert y \xrightarrow{c} x'}$	R.||8	$\dfrac{x \xrightarrow{a} \surd;\ y \xrightarrow{b} \surd;\ a\|b=c}{x \Vert y \xrightarrow{c} \surd}$
R.encaps1	$\dfrac{x \xrightarrow{a} x';\ a \notin H}{\mathbf{encaps}(H,x) \xrightarrow{a} \mathbf{encaps}(H,x')}$	R.encaps2	$\dfrac{x \xrightarrow{a} \surd;\ a \notin H}{\mathbf{encaps}(H,x) \xrightarrow{a} \surd}$
R.hide1	$\dfrac{x \xrightarrow{a} x';\ a \in I}{\mathbf{hide}(I,x) \xrightarrow{skip} \mathbf{hide}(I,x')}$	R.hide2	$\dfrac{x \xrightarrow{a} \surd;\ a \in I}{\mathbf{hide}(I,x) \xrightarrow{skip} \surd}$
R.hide3	$\dfrac{x \xrightarrow{a} x';\ a \notin I}{\mathbf{hide}(I,x) \xrightarrow{a} \mathbf{hide}(I,x')}$	R.hide4	$\dfrac{x \xrightarrow{a} \surd;\ a \notin I}{\mathbf{hide}(I,x) \xrightarrow{a} \surd}$
R.rec1	$\dfrac{\underline{u} \in \underline{D};\ y(\underline{u}) \xrightarrow{a} y';\ X(\underline{u}) = y(\underline{u})}{X(\underline{u}) \xrightarrow{a} y'}$	R.rec2	$\dfrac{\underline{u} \in \underline{D};\ y(\underline{u}) \xrightarrow{a} \surd;\ X(\underline{u}) = y(\underline{u})}{X(\underline{u}) \xrightarrow{a} \surd}$
R.sum1	$\dfrac{u \in D;\ x(u) \xrightarrow{a} x'}{\mathbf{sum}(d \text{ in } D,\ x(d)) \xrightarrow{a} x'}$	R.sum2	$\dfrac{u \in D;\ x(u) \xrightarrow{a} \surd}{\mathbf{sum}(d \text{ in } D,\ x(d)) \xrightarrow{a} \surd}$

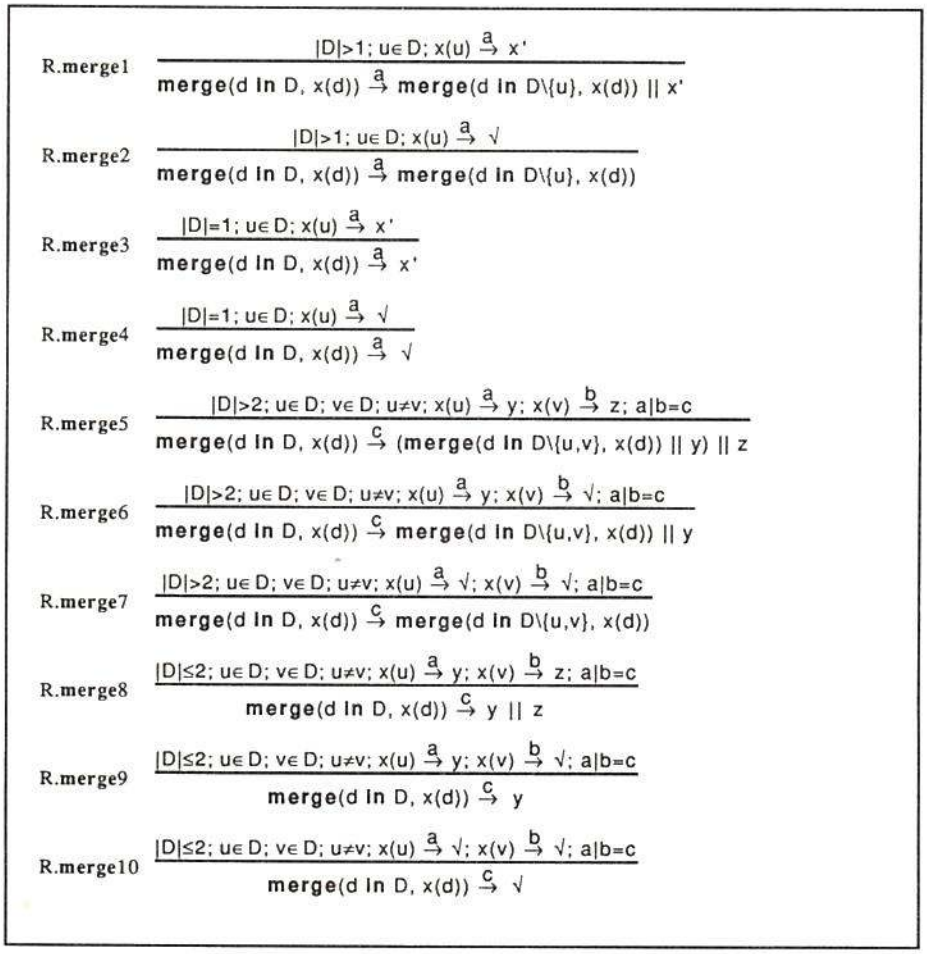

figure 3.5 Table of action relations.

3.7.4 Process Semantics

Now that we have defined the action relations for PSF_d we are able to assign a semantics to processes. In this case we define *bisimulation* [Par81] on top of these action relations.

A bisimulation is a binary relation R on process expressions, satisfying:

- if pRq and $p \xrightarrow{a} p'$, then $\exists q'$: $q \xrightarrow{a} q'$ and $p'Rq'$ ($[a] \in IA$)
- if pRq and $q \xrightarrow{a} q'$, then $\exists p'$: $p \xrightarrow{a} p'$ and $p'Rq'$ ($[a] \in IA$)
- if pRq then $p \xrightarrow{a} \surd$, if and only if $q \xrightarrow{a} \surd$ ($[a] \in IA$)

If there exists a bisimulation R on process expressions with pRq, then p and q are called *bisimilar*, notation $p \leftrightarroweq q$.

$\leftrightarroweq$ is a *congruence* on process expressions. See [BG87] for a proof.

3.7.5 An Example

Suppose we have the following definition of a certain process X:

```
X = sum(d in NAT, sum(e in BOOL, r(d,e)·Y(d,not(e))))
Y(d,true) = s(d)·s(true)·X
Y(d,false) = s(false)·s(d)·X
```

figure 3.6 Definition of process X.

Now we want to know what actions process X can perform. See the following example for the derivation of :

- $X \xrightarrow{r(5,false)} Y(5,not(false)) \xrightarrow{s(5)} s(not(false))\cdot X \xrightarrow{s(true)} X$

$$\textbf{R.a:}\ r(5,false) \xrightarrow{r(5,false)} \surd$$

$$\textbf{R.}\cdot\textbf{2:}\ \frac{r(5,false) \xrightarrow{r(5,false)} \surd}{r(5,false)\cdot Y(5,not(false)) \xrightarrow{r(5,false)} Y(5,not(false))}$$

$$\textbf{R.sum1:}\ \frac{r(5,false)\cdot Y(5,not(false)) \xrightarrow{r(5,false)} Y(5,not(false))}{\textbf{sum}(e\ \textbf{in}\ BOOL,\ r(5,e)\cdot Y(5,not(e))) \xrightarrow{r(5,false)} Y(5,not(false))}$$

$$\textbf{R.sum1:}\ \frac{\textbf{sum}(e\ \textbf{in}\ BOOL,\ r(5,e)\cdot Y(5,not(e))) \xrightarrow{r(5,false)} Y(5,not(false))}{\textbf{sum}(d\ \textbf{in}\ NAT,\ \textbf{sum}(e\ \textbf{in}\ BOOL,\ r(d,e)\cdot Y(d,not(e)))) \xrightarrow{r(5,false)} Y(5,not(false))}$$

$$\textbf{R.rec1:}\ \frac{X = \textbf{sum}(d\ \textbf{in}\ NAT,\ \textbf{sum}(e\ \textbf{in}\ BOOL,\ r(d,e)\cdot Y(d,not(e)))); \quad \textbf{sum}(d\ \textbf{in}\ NAT,\ \textbf{sum}(e\ \textbf{in}\ BOOL,\ r(d,e)\cdot Y(d,not(e)))) \xrightarrow{r(5,false)} Y(5,not(false))}{X \xrightarrow{r(5,false)} Y(5,not(false))}$$

figure 3.7 The derivation of a transition of X.

Now we have proved that it is possible to have a transition labeled with the atomic action *r(5,false)* from X to *Y(5,not(false))*. The next step is to show a possible atomic action to be performed by *Y(5,not(false))*.

$$\textbf{R.a:}\quad s(5) \xrightarrow{s(5)} \surd$$

$$\textbf{R.}\cdot\textbf{2:}\quad \frac{s(5) \xrightarrow{s(5)} \surd}{s(5)\cdot s(\text{not}(\text{false})) \xrightarrow{s(5)} s(\text{not}(\text{false}))}$$

$$\textbf{R.}\cdot\textbf{1:}\quad \frac{s(5)\cdot s(\text{not}(\text{false})) \xrightarrow{s(5)} s(\text{not}(\text{false}))}{s(5)\cdot s(\text{not}(\text{false}))\cdot X \xrightarrow{s(5)} s(\text{not}(\text{false}))\cdot X}$$

$$\textbf{R.rec1:}\quad \frac{s(5)\cdot s(\text{not}(\text{false}))\cdot X \xrightarrow{s(5)} s(\text{not}(\text{false}))\cdot X \;;\; Y(d,\text{not}(\text{false})) = s(d)\cdot s(\text{not}(\text{false}))\cdot X}{Y(5,\text{not}(\text{false})) \xrightarrow{s(5)} s(\text{not}(\text{false}))\cdot X}$$

figure 3.8 The derivation of a transition of X.

$$\textbf{R.a:}\quad s(\text{not}(\text{false})) \xrightarrow{s(\text{true})} \surd$$

$$\textbf{R.}\cdot\textbf{2:}\quad \frac{s(\text{not}(\text{false})) \xrightarrow{s(\text{true})} \surd}{s(\text{not}(\text{false}))\cdot X \xrightarrow{s(\text{true})} X}$$

figure 3.9 The derivation of a transition of X.

3.8 OTHER PROCESS SEMANTICS

In the previous section we have defined an operational semantics for process-expressions by means of action relations. These action relations are suitable as a base for the development of simulation tools. It can be used to define a semantic domain, i.e. the *graph model*, on which most of the known equivalence relations on processes can be defined.

We assign a graph to each process expression.

- Such a graph is *rooted* (i.e. there is just one root node)
- Each node is labeled with a closed process-expression possibly containing elements $[t_1],\ldots,[t_n]$ of the initial algebra of the data types.
- Each edge is labeled with elements $[a]$ of the initial algebra of atomic actions or *skip*.

Before we are able to define the graph of a certain process x, we have to define the set of all subprocesses of x. The definition of this set $Sub(x)$ is done recursively.

- $x \in Sub(x)$

- if $y \in Sub(y)$ and $y \xrightarrow{a} z$ can be derived from the action relations (for any a), then $z \in Sub(x)$

The graph of a certain process-expression x is constructed as follows:

- For each element y of $Sub(x)$, we generate a node labeled with y.
  Moreover there is one node that will be used as a terminal node, labeled with $\surd$.
- The node labeled with x is the root of the graph we are looking for.
- Next we add an edge, labeled with a, from node p to node q, whenever the corresponding transition $p \xrightarrow{a} q$ can be derived from the action relations.

Now that we have given a way of constructing graphs, it is possible to define a wide variety of semantics on this graph domain. These semantics include for example: *trace semantics, failure semantics* etc. [BKO87,BBK87]. We can also define *observational congruence* [Mil80] on this graph domain, which is in fact equal to our bisimulation semantics as defined in section 3.7.3.

4 EXAMPLES

In this section we give two examples of a specification in PSF_d, which illustrate the use of simple data types, process definitions and the concept of parameterization. The examples deal with a landing control system for an airport and the alternating bit protocol .

4.1 A LANDING CONTROL SYSTEM

4.1.1 The Problem

In the first example we specify a hypothetical landing control system for an airport. It is designed to handle the landing of a number of airplanes on a number of landing strips. Since the actual names of the airplanes and the strips can be considered as conditions local to some specific airport, we specify a control system which is parameterized with these items. The system consists of a number of parallel operating subsystems, first of which is the *Distribution* process. The other processes, the *Strip-Controllers*, all have the same behaviour. Each of them has control over exactly one landing strip.

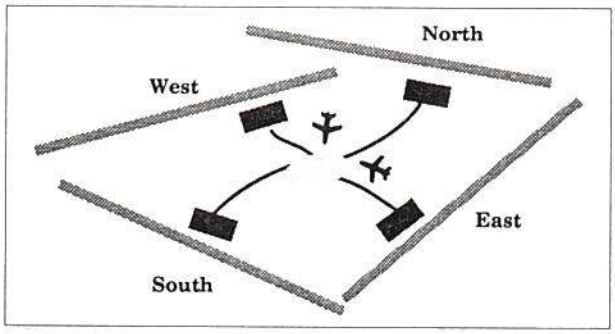

figure 4.1 Timbuktu Airport.

4.1.2 The Implementation

The process module *Landing-Control* has a parameter *Airport-Conditions*, which consists of the two sorts *STRIPS*, containing the names of the landing strips, and *PLANE-IDS*, containing the id's of all planes potentially willing to land. The module exports an atom *receive-req-to-land*, which enables the system to communicate with arriving airplanes, and the process *Control*, which is the name of the overall process being specified. Internal to this module are a number of atomic actions. The atoms *read*, *send* and *communicate* are used to model the communication between the process *Distribution* and each of the *Strip-Controllers*. The *STRIPS* argument determines which *Strip-Controller* is involved, and the *PLANE-IDS* argument indicates the plane that should be landed. As is indicated in the communications section, placing the atoms *send* and *read* in parallel yields the atom *communicate*. The set *H*, containing the *read* and *send* actions will be used to encapsulate unsuccessful communication. This happens when the *read* and *send* actions do not have a partner to communicate with. The other atomic actions, *land* and *disembark*, are not intended to take part in a communication.

Apart from the *Control* process we define three processes. The process *Distribution* receives a request to land from some plane and sends its id to one of the *Strip-Controllers*, which is willing to communicate with the *Distribution*. After that, the *Distribution* process starts all over again. The process *Strip-Control* is indexed with the name of some *STRIP*. In fact it defines a new process for each *STRIP*. It starts by receiving a message from the *Distribution* to handle a plane with a given id. After handling this plane, as defined by the process *Handle*, the *Strip-Controller* starts all over and is again able to receive a plane-id. The process *Handle* serves as a sub-process of the process *Strip-Control*. The second argument determines the plane and the first one determines the *STRIP* the plane must land on. This process stops after landing and disembarking the plane.

Finally the overall process *Control* is defined as the concurrent operation of the *Distribution* and all *Strip-Controllers*. The encapsulation operator removes unsuccessful communications.

4.1.3 The Specification

```
process module Landing-Control
begin

   parameters
      Airport-Conditions
          begin
             sorts
               STRIPS, PLANE-IDS
          end Airport-Conditions

   exports
      begin
          atoms
             receive-req-to-land : PLANE-IDS
          processes
             Control
      end
```

```
  atoms
     read, send, communicate : STRIPS # PLANE-IDS
     land                    : STRIPS # PLANE-IDS
     disembark               : PLANE-IDS

  processes
     Distribution
     Strip-Control : STRIPS
     Handle        : STRIPS # PLANE-IDS

  sets
     of atoms
        H = {read(s,id), send(s,id) | s in STRIPS, id in PLANE-IDS }

  communications
     send(s,id) | read(s,id) = communicate(s,id)
        for s in STRIPS, id in PLANE-IDS

  variables
     s :-> STRIPS
     id :-> PLANE-IDS

  definitions
     Distribution = sum(id in PLANE-IDS, receive-req-to-land(id).
                                  sum(s in STRIPS, send(s,id))
                    ) . Distribution
     Strip-Control(s) = sum(id in PLANE-IDS, read(s,id).Handle(s,id)
                        ) . Strip-Control(s)
     Handle(s,id) = land(s,id) . disembark(id)
     Control = encaps(H, Distribution ||
                         merge(s in STRIPS, Strip-Control(s)))

end Landing-Control
```

figure 4.2 Specification of a generic landing control system.

This specification can be used as a generic specification for *Landing-Controllers*. A *Landing-Control* at for instance *Timbuktu-Airport* can be constructed by binding a module which defines the landing strips and the planes that potentially land at *Timbuktu-Airport* to the parameter of *Landing-Control*. A graphical representation is given in figure 4.4.

```
data module Timbuktu-Airport
begin

   exports
      begin
         sorts
            Timbuktu-STRIPS, Timbuktu-PLANE-IDS
         functions
            North : -> Timbuktu-STRIPS
            East  : -> Timbuktu-STRIPS
            South : -> Timbuktu-STRIPS
            West  : -> Timbuktu-STRIPS
            KL204 : -> Timbuktu-PLANE-IDS
            SQ001 : -> Timbuktu-PLANE-IDS
            JL403 : -> Timbuktu-PLANE-IDS
            PA666 : -> Timbuktu-PLANE-IDS
            HA345 : -> Timbuktu-PLANE-IDS
      end

end Timbuktu-Airport
```

```
process module Timbuktu-Landing-Control
begin
   imports

      Landing-Control [Airport-Conditions bound by
                       [STRIPS -> Timbuktu-STRIPS,
                        PLANE-IDS -> Timbuktu-PLANE-IDS ]
                      to Timbuktu-Airport ]

end Timbuktu-Landing-Control
```

figure 4.3 Timbuktu Airport definition.

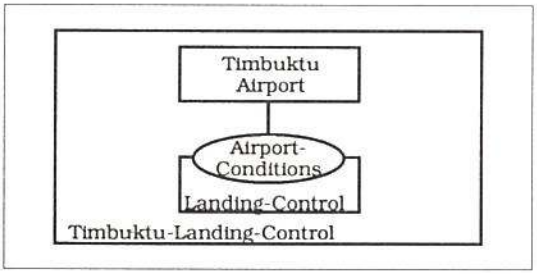

figure 4.4 Timbuktu Airport structure diagram.

4.2 ALTERNATING BIT PROTOCOL

4.2.1 The Problem

One of the most famous communication protocols is the Alternating Bit Protocol (ABP). It has been used many times to serve as a test case for a new formalism. Our specification emanates from the ABP specification in ACP as described in [BK86a,BK86b].

We can represent the Alternating Bit Protocol with a picture as follows:

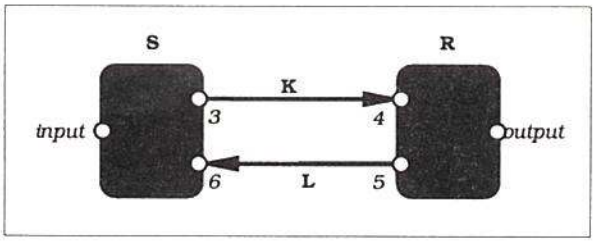

figure 4.5 Graphical representation of the Alternating Bit Protocol.

It consists of four components:

- S : The sender.
- R : The receiver.

- K : A channel connecting the sender and the receiver.
- L : A channel connecting the receiver and the sender.

The goal of the Alternating Bit Protocol is to transport data items from a certain set D from the input port to the output port. In the next paragraphs we give a description of each component.

4.2.1.1 The Sender

First, component S reads a message at the input port. This message is extended with a *control boolean* to form a so-called *frame* and this frame is sent along channel K. The sending of the frame proceeds until component S receives an acknowledgement of a successful transmission at channel L. After a successful transmission component S flips the control boolean and starts all over again.

4.2.1.2 Communication Channel *K*

Component K transmits frames from the sender to the receiver. There are two situations that can occur when sending information along channel K.

- The frame is properly transmitted.
- The frame is corrupted during the transmission.

We assume channel K to be *fair*, i.e, it will not produce an infinite stream of corrupted data.

4.2.1.3 The Receiver

The receiver R reads a frame from channel K. We assume that R is able to tell, e.g. by performing a *checksum control*, whether or not the frame has been corrupted. When the frame is correct R checks the control boolean in the frame. If this control boolean matches the internal control boolean of K, the message in the frame is sent to the output port, K flips its internal boolean and starts waiting for the next frame to arrive. In all other cases R sends the complement of its own control boolean along channel L and waits for the retransmission of the frame.

4.2.1.4 Communication Channel *L*

Component L is used to transmit *receive acknowledgements* from the receiver to the sender. Like channel K, channel L is able to corrupt data. We will assume that the sender S can tell whether an acknowledgement has been corrupted. We assume that channel L is fair too.

4.2.2 The Specification

```
data module Booleans

begin
   exports
      begin
         sorts
           BOOLEAN
         functions
           true  :                          ->   BOOLEAN
           false :                          ->   BOOLEAN
           and   :   BOOLEAN # BOOLEAN      ->   BOOLEAN
           or    :   BOOLEAN # BOOLEAN      ->   BOOLEAN
           not   :   BOOLEAN                ->   BOOLEAN
      end
```

```
    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


data module ABP-Ports
begin

   exports
      begin
         sorts
           ABP-PORT
         functions
           p3 : -> ABP-PORT
           p4 : -> ABP-PORT
           p5 : -> ABP-PORT
           p6 : -> ABP-PORT
      end

end ABP-Ports


data module Errors
begin

   exports
      begin
         sorts
           ERROR
         functions
           ce : -> ERROR
      end

end Errors


data module Bits
begin

   exports
      begin
         sorts
           BIT
         functions
           0 : -> BIT
           1 : -> BIT
      end

end Bits
```

```
process module Producer
begin

   parameters
      Ext-Ports
         begin
            atoms
               output : BIT
         end Ext-Ports

   exports
      begin
         processes
            PROD
      end

   imports
      Bits

   definitions

      PROD = (skip . output(0) + skip . output(1)) . PROD

end Producer


process module Consumer
begin

   parameters
      Data-Items
         begin
            sorts
               DATA
         end Data-Items,

      Ext-Ports
         begin
            atoms
               input : DATA
         end Ext-Ports

   exports
      begin
         processes
            CONS
      end

   definitions

      CONS = sum(d in DATA, input(d)) . CONS

end Consumer


process module ABP
begin

   parameters
      Data-Items
         begin
            sorts
               DATA
         end Data-Items,
```

```
   Ext-Ports
      begin
         atoms
            input : DATA
            output : DATA
      end Ext-Ports

exports
   begin
      processes
        ABP
   end

imports
   Booleans, ABP-Ports, Errors

atoms
   r,s,c : ABP-PORT # DATA # BOOLEAN
   r,s,c : ABP-PORT # BOOLEAN
   r,s,c : ABP-PORT # ERROR

processes
   S,K,L,R
   RM            : BOOLEAN
   SF,RA,K,SM    : DATA # BOOLEAN
   L,RF,SA       : BOOLEAN

sets
   of ABP-PORT
      FRAME-PORT = {p3,p4}
      ACK-PORT = {p5,p6}
      ERROR-PORT = {p4,p6}

   of atoms
      H = { s(p,d,b), r(p,d,b) | p in FRAME-PORT, d in DATA,
                                                  b in BOOLEAN } +
            { s(p,b), r(p,b) | p in ACK-PORT, b in BOOLEAN } +
              { s(p,e), r(p,e) | p in ERROR-PORT, e in ERROR }

communications
   s(p,d,b) | r(p,d,b) = c(p,d,b) for p in FRAME-PORT, d in DATA,
                                                  b in BOOLEAN
   s(p,b) | r(p,b) = c(p,b) for p in ACK-PORT, b in BOOLEAN
   s(p,e) | r(p,e) = c(p,e) for p in ERROR-PORT, e in ERROR

variables
   b : -> BOOLEAN
   d : -> DATA


definitions

   S = RM(false)
   RM(b) = sum(d in DATA, input(d) . SF(d,b))
   SF(d,b) = s(p3,d,b) . RA(d,b)
   RA(d,b) = (r(p6,not(b)) + r(p6,ce)) . SF(d,b) + r(p6,b) . RM(not(b))

   K = sum(d in DATA, sum(b in BOOLEAN, r(p3,d,b))) . K(d,b)
   K(d,b) = (skip . s(p4,ce) + skip . s(p4,d,b)) . K

   R = RF(false)
   RF(b) = sum(d in DATA, r(p4,d,not(b)) + r(p4,ce)) . SA(not(b)) +
           sum(d in DATA, r(p4,d,b) . SM(d,b))
   SA(b) = s(p5,b) . RF(not(b))
   SM(d,b) = output(d) . SA(b)
```

```
      L = sum(b in BOOLEAN, r(p5,b) . L(b))
      L(b) = (skip . s(p6,ce) + skip . s(p6,b)) . L

      ABP = encaps(H, S || K || R || L)

end ABP


process module Communication-Ports
begin

   parameters
      Data-Items
         begin
            sorts
              DATA
         end Data-Items

   exports
      begin
         atoms
          prod-out,abp-in,abp-out,cons-in,prod-abp-comm,abp-cons-comm : DATA

         sets
          of atoms
            H = { prod-out(d),abp-in(d),abp-out(d),cons-in(d) | d in DATA }
      end


   communications
      prod-out(d) | abp-in(d) = prod-abp-comm(d) for d in DATA
      abp-out(d) | cons-in(d) = abp-cons-comm(d) for d in DATA

end Communication-Ports


process module System-Ports
begin

   imports
      Communication-Ports { Data-Items bound by
                            [ DATA -> BIT ] to Bits }
end System-Ports


process module System
begin

   exports
      begin
         processes
           SYS
      end

   imports
      Producer { Ext-Ports bound by
                 [ output -> prod-out ] to System-Ports },
      Consumer { Data-Items bound by
                 [ DATA -> BIT ] to Bits
                 Ext-Ports bound by
                 [ input -> cons-in ] to System-Ports },
      ABP { Data-Items bound by
            [ DATA -> BIT ] to Bits
            Ext-Ports bound by
            [ input -> abp-in,
              output -> abp-out ] to System-Ports }
```

```
    definitions
      SYS = encaps(H, ( PROD || ABP || CONS ))

end System
```

figure 4.6 PSF_d specification of the Alternating Bit Protocol.

In this solution the module that is dealing with the Alternating Bit Protocol, is part of a bigg system that also contains a producer of (random) data elements and a consumer. Th interconnection of modules is established by communications as defined in *System-Port*. This solution is an example of how modularization and parameterization is achieved in PSF. To point out the constitution of module *System*, figure 4.7 shows the visualization of th imports, at the top level.

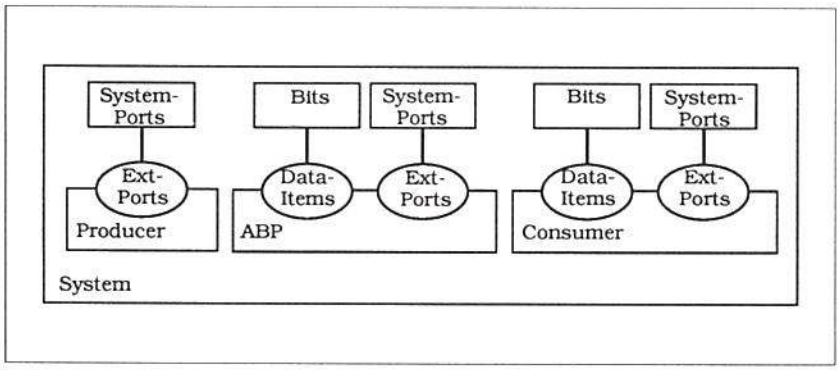

figure 4.7 Structure diagram of the ABP specification.

5 Considerations & Comparisons

5.1 Implementation of PSF_d by means of SDF

As mentioned earlier it is possible to construct a parser and a syntax directed editor from th syntax definition of a language in SDF. We have constructed a prototype syntax checker and syntax directed editor by using these possibilities. The syntax of the examples in this report have actually been checked by means of these programs. There are, however, two reasons not to go on in this way and extend the parser to incorporate semantic information in order to make a type-checker. The first reason is that the research on automatically implementing SDF specifications is still in progress and the second reason is that at this moment the performance, as far as the speed of execution is concerned, of the parser and the editor is rather poor.

5.2 A COMPARISON WITH OTHER FORMAL DESCRIPTION TECHNIQUES

In this section we will compare PSF_d and some other Formal Description Techniques. We will however focus on the comparison with LOTOS.

5.2.1 LOTOS

LOTOS (Language of Temporal Ordering Specification) is one of the two Formal Description Techniques, developed within ISO (International Organization for Standardization) for the formal specification of open distributed systems, in particular for those related to the Open Systems Interconnection (OSI) computer network architecture.

5.2.1.1 Similarities

Like PSF_d, LOTOS is a combination of two formalisms, namely a variation on ACT ONE [EM85] to describe data types and a process description part based on CCS [Mil80]. As opposed to PSF_d , which was designed to be as close to ACP as possible, the distance between LOTOS and CCS is much greater. Many differences between LOTOS and PSF_d originate from the differences between ACT ONE and ASF, and CCS and ACP. We will start off with a list of constructions that are available in both languages:

LOTOS	*PSF_d*	
i	skip	
B1 [] *B2*	*B1* + *B2*	
choice *x*:*D* [] *B*(*x*)	sum(*x* in *D*, *B*(*x*))	
par *g* in [g_1, ..., g_n] <*parallel-op*> *B*	merge(*g* in *G*, *B*)	where $G = \{g_1, ..., g_n\}$
hide g_1, ..., g_n in *B*	hide(*G*, *B*)	where $G = \{g_1, ..., g_n\}$

figure 5.1 Similarities between LOTOS and PSF_d.

From this table it is clear that in LOTOS you have to specify a set of *gates* in the *hide* and *par* operation by summing up all elements, whereas in PSF_d it is possible to construct such a set with more powerful operators and subsequently attach a name to it.

5.2.1.2 Action Prefix vs. Sequential Composition

One of the major differences between LOTOS and PSF_d is the way in which sequential composition is expressed. In ACP processes can be linked together by means of the · -operator. CCS, however, only considers *action prefix*. This means that it is only possible to put an atomic action in front of a process or *behaviour expression*. In order to have a sequential composition on behaviour expression a new operator, the *enable* operator, had to be introduced.

LOTOS	*PSF_d*
g; *B*	*g* · *B*
B1 » *B2*	*B1* · *B2*

figure 5.2 Action prefix vs. sequential composition.

5.2.1.3 Concurrency

Yet another difference occurs when expressing that processes have to be executed concurrently. In LOTOS there are three operators to express concurrency.

- *B1* |[$g_1, \ldots, g_n$]| *B2*
  This is the most general operator. It states that two processes *B1* and *B2* have to synchronize at gates $g_1, \ldots, g_n$.
- *B1* || *B2*
  The actions from *B1* and *B2* have to synchronize in each step.
- *B1* ||| *B2*
  There is no synchronization between *B1* and *B2* at all. This is called *interleaving*

Synchronization means that both processes have to be willing to execute a g, from the given set, simultaneously. So in LOTOS synchronization is only possible between identical actions as opposed to PSF_d where communication is settled by the definition of a communication function which leads to a more general concept of communication. To force communication in PSF_d the *encaps* operator is used.

LOTOS	*PSF_d*
B1 |[$g_1, \ldots, g_n$]| *B2*	encaps(G, *B1* || *B2*) where $G = \{g_1^*, \ldots, g_n^*, g_1\hat{}, \ldots, g_n\hat{}\}$ $g_i^* \mid g_i\hat{} = g_i$
B1 || *B2*	encaps(A, *B1* || *B2*) where $A = \{a_1^*, \ldots, a_n^*, a_1\hat{}, \ldots, a_n\hat{}\}$ $a_i^* \mid a_i\hat{} = a_i$ for all atomic actions in the alphabets of *B1* and *B2*.
B1 ||| *B2*	*B1* || *B2*

figure 5.3 Concurrency constructs.

5.2.1.4 Communication

In LOTOS all communication takes place at *gates*. We have already shown that an action/gate can synchronize with an identical action/gate. However, it is also possible to transfer data from one process to another by means of synchronization. This is achieved by two constructions:

- *value declaration* : !E, where E is a *value expression*, i.e. a LOTOS expression describing a data value.
  examples: !TRUE, !(3+5), !(x+1), !'example', !min(x,y)
- *variable declaration* : ?x:t, where x stands for a variable of type t.
  examples: ?x:integer, ?switch:boolean

A gate can be coupled with one of these constructions so that the expression $g?x{:}integer$ describes the set of all actions $g\langle v\rangle$ where v is an instance of sort *integer*.

LOTOS	PSF_d
g ! *a*(*y*)	*send*(*a*(*y*))
g ? *x*:*t*	sum(*x* in *t*, *receive*(*x*))

figure 5.4 Communication constructs.

5.2.1.5 Features supported by LOTOS but not by PSF_d

There are some features in LOTOS that PSF_d (currently) does not support. Two of these features use *conditional constructs*. Conditional constructs are expressed as an equation between two value expressions or boolean expressions. In the former case, the condition is met if the two expressions evaluate to exactly the same value. Conditional constructs are used in synchronization as a *selection predicate* to impose a restriction on the values that may be transferred, and as *guards* in *guarded expressions*. PSF_d does not support conditional constructs, but has nevertheless the same expressive power. This is achieved by using *sets*, however we admit that this is carried out in a rather cumbersome way.

LOTOS	PSF_d
g ? *x*:*integer*[*x*<3]	sum(*x* in *I*, *receive*(*x*))
	I is the set representing the integers smaller than 3.
[*x* > 3] → *process1*	*X*(*g*) = *process1* where $g \in$ Int / { 0, ..., 3}
[] [*x* = 5] → *process2*	*X*(5) = *process2*
[] [*x* < 9] → *process3*	*X*(*l*) = *process3* where $l \in$ { 0, ..., 9}

figure 5.5 Conditional constructs and their translations in PSF_d.

Another feature that is not present in PSF_d is the *disabling* operator. The LOTOS expression: *1* [> *B2*, means that as long as *B1* is active *B2* can take over the execution at any time, resulting in the disappearance of *B1*. This operator has just recently been introduced in ACP [Ber88].

In LOTOS each behaviour expression has a *functionality*. This functionality is used whenever one process, upon successful termination, enables another process and wants to send some data to the enabled process. When combining behaviour expressions by means of an operator, the functionality of the total expression depends on the functionality of the operands. There are three main types of functionality:

- noexit no successful termination. Deadlock or explicit *stop*.
- exit successful termination.
- $E_1, \ldots, E_n$ a list of value expressions. Successful termination with value passing.

In PSF_d there is no such thing as value passing. All processes coexist and exchange information by communication, although a lot of them may be held up, waiting to take part in a communication. The *chaining* operator, which is merely an abbreviation of two renamings, a merge and an abstraction, in ACP [Vaa86b] resembles the enabling operator but has a slightly different semantics.

5.2.1.6 Data Specification

As stated earlier LOTOS uses a variation on ACT ONE for the specification of the data types. Though the syntax of the data specification parts of LOTOS and PSF_d differs, they look very much like each other. This includes parameterization and renaming of imported sorts and functions. The only difference is, that it is not possible to define a hidden signature in LOTOS. This would be like defining all sorts and all functions in the *exports* section in PSF_d.

5.2.1.7 Modularization

Though modularization is possible when defining data types, LOTOS does not support such a powerful concept of importing and exporting process definitions. We think of this as a serious shortcoming in LOTOS. The only way to have some abstraction is by writing a specification in a stringent top-down manner using the *where* construction. An example 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
```

figure 5.6 Example of a LOTOS specification.

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.

5.2.2 Estelle

Estelle [ISO86] is the other Formal Description Technique developed by the ISO. Estelle is based upon an extended finite state machine model.

Finite state machines are a class of theoretical automata and have been widely used in the field of compiler design for string recognition in the lexical analysis and parsing of programming languages. Finite state machines are often depicted by graphs with the nodes representing states and the edges representing a transition from one state to another. The labels connected to the edges identify the input that causes the transition from one node to another.

A specification in Estelle consists of a set of modules which can communicate with each other. Modules represent finite state machines and are defined by using a number of primitives which are extensions to ISO Pascal. So a specification in Estelle looks like a Pascal program with some extra facilities. Being based on Pascal, there are no abstract data types and verification of specifications is hindered.

5.2.3 COLD

COLD [FJKR87] is a series of languages developed in the framework of ESPRIT project 432 (METEOR). COLD is defined by means of a translation of its grammatical constructs to the constructs of a three layered formal language. The top layer of this kernel is a special version of lambda calculus, which is called $\lambda\pi$, and is used for modelling parameterization. Expressions in this lambda calculus contain terms from a special many-sorted algebra, called CA, which is used for modelling modularization constructs. This algebra constitutes the middle layer. The constants used in the terms of this algebra are presentations of logical theories. The logical language used at the bottom level is based on a special infinitary logic, called MPL_{ω}. Every construct in a COLD specification corresponds with an expression in the kernel of formal languages with a well-defined semantics. COLD specifications are translated by means of attribute grammars to the kernel.

COLD focuses very strongly on the mathematical basis of the language, which guarantees a nice framework for verification. The current version of COLD, i.e. COLD-K, does not support concurrency. Research in this area is currently being carried out.

5.2.4 SMoLCS

SMoLCS is an integrated methodology for the specification of concurrent systems and languages developed mainly by Astesiano and Reggio in cooperation with Wirsing. In [AR87] SMoLCS-driven concurrent calculi are presented. The differences with PSF_d are the following:

- in the SMoLCS approach the processes are represented as special instances of data types, notably transition systems, as opposed to PSF_d where there are special constructs in the language to model processes.
- SMoLCS is biased towards CCS rather than ACP.
- in the SMoLCS approach partial data types are used, whereas PSF_d uses total functions only. As a result definedness predicates are used in the former formalism and so it has more in common with COLD than with PSF_d as far as the data type section is concerned.

5.2.5 FOREST

FOREST is a specification language that has been developed at the Imperial College in London by a team around Maibaum, see GOLDSACK [G88]. The language uses deontic logic to express (potential) system behaviour. The behaviour of agents is formalized in terms of modal action logic. The data are described in terms of a first order language based on the declaration of structured signatures. The semantics of the agents is given in the context of trace theory. The formalism FOREST provides a combination of data type specifications and process (agent) specifications just as PSF_d does. The main difference is that FOREST uses a process logic, whereas PSF_d uses a process algebra.

5.3 A COMPARISON WITH PARALLEL PROGRAMMING LANGUAGES

There are some programming languages that allow the writing of concurrent programs. In this section we will discuss some of these programming languages and compare them with PSF_d.

5.3.1 Extended programming languages

By extended programming languages we mean languages that have been extended afterwards to include features to support concurrent programming. Two examples of these languages are *Concurrent Pascal* [Bri75] and *Concurrent Euclid* [Hol83]. Both languages have some extra features like *processes* to define concurrently executable pieces of the program and *monitors* to guarantee (mutual) exclusive access to variables. Communication between processes is established by means of shared variables.

5.3.2 Modula, CHILL, Ada

Modula, CHILL and Ada are all based on Pascal. Though they have been designed to be able to deal with concurrent programming, they use essentially the same constructs as the languages from the previous section. All three languages allow the description of sequential pieces of program that can be executed concurrently. In Modula and CHILL these constructs are called: processes and in Ada: *tasks*. There are some different solutions to the inter-process communication.

In Modula communication between two processes is either established by sharing data through an *interface module* (=monitor), or by synchronization. Synchronization means that one process waits until another process has reached a certain state. This is achieved by the sending of and waiting for *signals*.

In CHILL there are three ways for processes to communicate with each other. The first way is by means of *regions*, which can be compared with Modula's interface modules. Next come the *buffers* which operate like some kind of mailbox in which one process leaves a message of a certain type that can be picked up by another process. The last way of communicating is by means of *signals*. Signals can directly be sent from one process to another process and it is possible to specify to which processes a signal is restricted. Any intermediate buffering is taken care of by the underlying system.

In Ada there is only one way in which two tasks can communicate, the *rendez-vous*. A task that wants to communicate with another task, starts waiting until the other task wants to communicate too. When both are willing to communicate they exchange information and go on with the execution of their own instructions.

The rendez-vous communication resembles the communication in PSF_d the most. In PSF_d a process is not able to proceed until the other party wants to perform the complementary communication action. There is however still a difference. In Ada there is an asymmetry in the communication. There is one task that *accepts* a communication and as such controls the communication. The other task does an *entry call* to this specific task. Whenever two tasks are willing to communicate, the called task executes the sequence of statements of the *accept* statement while the calling task remains suspended. Such an asymmetry does not exist in PSF_d where all processes are equal partners in communication and both supply one half of the communication. Moreover PSF_d processes do not state with which specific process they want to communicate. This is an advantage when constructing specifications in a bottom-up fashion.

A feature that is lacking in PSF_d but present in Ada is: *time*. It is possible in Ada to delay a process for a while by means of the *delay* statement. In the context of a *select* statement, that implements the idea of choosing non-deterministically between guarded commands (see

[Dijk76]), a *delay* develops into a *time-out*, when used as a *guard*. This means that only after a certain period of time one branch of a select statement becomes active, i.e. the guard becomes true, when all the guards of the other branches remain false. There is strong need for such mechanisms in real-time applications.

An important difference between PSF_d and the aforementioned languages is, that, being based on Pascal, these languages are all imperative languages and PSF_d is not. The architecture of conventional machines has influenced the development of programming languages tremendously. Three characteristics of imperative languages show this influence:

- Variables.

  A major component of a computer is the memory, which comprises a large number of *memory cells*. The reflection of these memory cells in a programming language are variables.

- Assignment Operation

  Closely tied to the memory architecture is the notion that everything that has been computed must be stored, i.e., assigned to a cell. This accounts for the assignment operation in imperative programming languages.

- Repetition

  A program in an imperative language usually accomplishes its task by executing a sequence of elementary steps repeatedly. The von Neumann architecture forces this way of solving problems because of the instructions that are stored in the memory.

In PSF_d we think of the execution of a large program as being merely a bunch of processes floating around and sometimes communicating with each other, having no relation with the architecture on which the program is executed.

All three languages mentioned, support some kind of *information hiding* by allowing the definition of *abstract data types*. This is achieved by defining a data type and some functions that operate on this type that are grouped together in some construct. The representation of the abstract data types and the implementation of the functions is defined within this construct, but is hidden for the outside world. In this way the outside world gets only an abstract view of the data types involved. Information hiding is provided by *modules* in Modula and CHILL and by *packages* in Ada. In PSF_d, data abstraction and procedural abstraction is provided by *data* and *process modules* and the *import* and *export* constructions. There are some differences however. In PSF_d an imported object automatically appears in the export section of the importing module. This is not the case in any of the three programming languages. There is neither something like the *origin rule* as in PSF_d, allowing multiple imports of the same module.

Another important feature in new programming languages is the *generic* module. A generic module can be looked upon as a template for a module, in which one or more types used in the module are parameterized. This can be used in, e.g., specifying a queue, for which it is possible to define the actions that can be performed on the data objects, though the type of the objects is not known in advance. Ada and PSF_d support generic modules, the latter by means of the *parameters* section, while Modula and CHILL do not. Two other features that both Ada and PSF_d provide, but CHILL and Modula lack, are overloading and renaming of objects.

5.3.3 POOL 2

POOL (Parallel Object-Oriented Language) [Ame88] is a programming language designed to integrate object oriented programming with parallelism. All objects in a POOL program may execute in parallel, so this resembles the notion of processes in PSF_d.

Each *object* is an instance of a *class* and can be looked upon as a process containing some internal data and some *methods* that can operate on these data. The internal information of an object cannot be accessed by other objects directly, but objects exchange information by sending *messages*. Upon receiving a message an object executes the appropriate method and the object that is the result of this *method execution* is sent back to the original sender. In this way information hiding and abstraction is achieved. This communication is again, like in Ada, asymmetrical.

Units consist of two parts; the *specification unit* and the *implementation unit*. A unit is the building block for modularization. An implementation unit consists of a set of class definitions. Which classes, from the implementation unit, are visible to the outside world is defined in the specification unit (cf. PSF_d's *exports*). In both the implementation unit and specification unit classes en methods from other units can be made visible by means of the *use* construct (cf. PSF_d's *import*). It is possible to have generic classes in POOL and renaming of *class names* and names of *globals* is possible. POOL supports no overloading.

In [Vra88] we can find a study of how to implement ACP specifications in POOL and a more extensive comparison between POOL and ACP.

5.3.4 Occam 2

Occam [INM88] is a programming language based on CSP [Hoa85]. It has been designed by INMOS and serves as a programming language for the INMOS transputer, which in turn can be considered an Occam machine. The transputer is a single processor with some internal memory and has four channels with which it can be connected to neighbouring transputers. It is expected that a set of such transputers will form an easily extendible parallel computer.

Occam, being closely related to the architecture of the transputer, is a rather low-level programming language. The data types are very simple: booleans, bytes, integers, reals and arrays of aforementioned. Characters are represented by bytes and strings by arrays of bytes. Occam allows no user-defined types and it has no modularization concepts like the *imports/exports* mechanism in PSF_d. It allows the construction of a larger process from three primitive processes:

- assignment
- input
- output

In constructing larger processes the programmer states which parts of the program may be executed in parallel and which parts must be executed sequentially. Communication of values between processes is achieved by *channels*. The format and data type of these values is specified by the *channel protocol*. The channel protocol may consist of a list of data types and consequently each communication along this channel must match this protocol exactly, both at the side of the sender as well as the receiver. Communication in Occam is again asymmetrical.

Occam does include one feature that is not present in PSF_d, namely the *timer*. A timer is some kind of clock that supplies integer values and is incremented at regular intervals. Again, such a feature is of course very important for real-time applications.

5.3.5 PARLOG

PARLOG [CG84] is a parallel logic programming language that is characterized by the use of the concepts of guards and committed choice nondeterminism as in Dijkstra's procedural language of guarded commands [Dijk76]. It is a member of a family of languages that further consists of Concurrent Prolog [Sha83] and GHC [Ued85].

PARLOG offers parallel evaluation of *and*- and *or*-clauses. Shared variables, used by clauses evaluated concurrently, act as communication channels. Both *synchronous* and *asynchronous communication* can be used by the programmer.

PARLOG does not provide any means for specifying abstract data types, all data types have to be represented by the programmer, by means of *lists*, nor does it incorporate any modularization concepts. PARLOG is of interest however, because it has been used to translate LOTOS specifications into an executable PARLOG program [Gil87].

5.4 FUTURE DEVELOPMENTS

As we stated in the introduction, PSF_d is the base for a set of tools to help in writing specifications in ACP. A lot of work has still to be carried out. In this section the plans for developments in the near future are presented.

5.4.1 Tools

At the moment work is carried out on the first step towards tools based on PSF_d. We have a syntax checker, which was built using a LEX and YACC front-end, and we have a prototype of a type checker. This type checker is based on the already existing tools for ASF.

When building a set of independent tools for the analysis of PSF_d specifications, an intermediate language of a very simple nature is needed to avoid that every tool has to process the PSF_d text again. This Tool Interface Language (TIL) should contain e.g. redundant type information of functions, no modularization constructs and should be easy to parse. Current research is carried out on defining TIL, and writing a compiler for PSF_d that produces TIL.

The first tool to be developed will be a simulator, which generates traces of a specified process. The next step is a tool for the verification of processes. Correctness of a specification is verified by proving it equivalent to some simpler specification. Several algorithms can be implemented for testing equivalence of processes. Since bisimulation in general is undecidable, for complex (i.e. not regular) processes tools should be developed to aid manual verification. A last step is an implementation tool, that will implement a specification in some kind of programming language, hopefully to be executed on a parallel computer.

5.4.2 The Language

Though PSF_d as presented in this report is already a rather powerful specification language, yet we are thinking of some enhancements. There are still some ACP constructions that have not been implemented in PSF_d. These constructions are, e.g.: chaining, (dynamic) process

creation, renaming (of atoms), interrupts, priorities and mode transfer. We will have to examine which constructions can be incorporated in PSF_d without affecting the semantical model as yet defined.

There is still one open problem that has to be resolved, namely the normalization, or *flattening*, of a PSF_d specification. Furthermore we have to investigate whether we would like to add some kind of *guarded process definitions* like the guarded commands in LOTOS.

Furthermore it is possible that some constructions currently available in PSF_d have to be redesigned to match future requirements. One of this constructions is the communication between atoms. In this version of PSF we have imposed three semantical constraints on the definition of communication. Due to the fact that we only consider communication satisfying *handshaking*, it might be possible that some of these restrictions can be dropped when using a different syntax.

It is also possible that one of the three main building blocks of PSF (data specification, process specification, modularization) is exchanged for another formalism in the future. It has been one of the design criteria for PSF_d to let this building blocks interfere with each other as little as possible, to guarantee interchangeability.

5.5 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, as opposed to the informal way in which data types are generally treated in ACP. We hope that PSF_d will be a contribution to the construction of more reliable software.

ACKNOWLEDGEMENTS

Thanks are due to Jos Baeten for intensive guidance, suggestions and careful reading of earlier drafts, to Jan Bergstra for proposing, as a subject of further investigation, the comparison between ACP and LOTOS, from which the design of PSF_d evolved, to Frits Vaandrager and Jos Vrancken for useful comments on several chapters of this report, to Jan Rekers for his assistance in implementing the SDF specifications, to Paul Klint for his suggestions concerning the implementation of PSF_d, to Wilco Koorn for implementing the syntax checker and all other people that have contributed to this paper, especially the participants of the process algebra meetings.

6 REFERENCES

[Ame88] P. America, *Definition of POOL 2, a parallel object-oriented language*, Esprit project 415A, Doc. Nr. 0364, Philips Research Laboratories, Eindhoven, 1988.

[AR87] E.Astesiano, G.Reggio, *SMoLCS-Driven Concurrent Calculi*, in: Proc. International Joint Conference on Theory and Practice of Software Development, TAPSOFT '87, (H. Ehrig, R. Kowalski, G. Levi, U. Montanari, eds.), LNCS 249, pp. 169-201, Springer Verlag, 1987.

[Bae86] J.C.M. Baeten, *Procesalgebra*, Kluwer, Deventer, 1986. *(in Dutch)*

[Bar82] J.G.P. Barnes, *Programming in Ada*, Addison-Wesley, 1982.

[BB87] J.C.M. Baeten, J.A. Bergstra, *Global Renaming Operators in Concrete Process Algebra (revised version)*, Information & Computation 78 (3), pp. 205 - 245, 1988.

[BB88] J.C.M. Baeten & J.A. Bergstra, *Processen en procesexpressies*, Informatie 30 (3), pp. 214-222, 1988. *(in Dutch)*

[BBK87] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Ready trace semantics for concrete process algebra with priority operator*, in: British Computer Journal 30 (6), pp. 498-506, 1987.

[Ber88] J.A. Bergstra, *A mode transfer operator in process algebra*, Report P8808, University of Amsterdam, 1988.

[BG87] J.C.M. Baeten & R.J. van Glabbeek, *Merge and termination in process algebra*, in: Proc. 7th Conf. on Foundations of Software Technology & Theoretical Computer Science, Pune, India (K.V. Nori, ed.), Springer LNCS 287, pp. 153-172, 1987.

[BHK85] J.A. Bergstra, J. Heering & P. Klint, *Algebraic definition of a simple programming language*, Report CS-R8504, Centre for Mathematics and Computer Science, Amsterdam, 1987.

[BHK89] J.A. Bergstra, J. Heering & P. Klint (eds.), *Algebraic Specification*, ACM Press Frontier Series, Addison-Wesley, 1989.

[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.

[BK86c] J.A. Bergstra & J.W. Klop, *Algebra of communicating processes*, in: Mathematics & Computer Science I (J.W. de Bakker, M. Hazewinkel & J.K. Lenstra, eds.), C.W.I. Monograph 1, North-Holland, Amsterdam, 1986.

[BKO87] J.A. Bergstra, J.W. Klop & E.-R. Olderog, *Readies and failures in the algebra of communicating processes*, SIAM J. of Comp., Vol. 17, No. 6, pp. 1134-1177, 1988.

[Bri75] P. Brinch Hansen, *The Programming Language Concurrent Pascal*, in: IEEE Transactions on Software Engineering, Volume SE-1, pp 199-207, 1975.

[BTL79] Bell Telephone Laboratories, *UNIX Programmer's Manual*, 1979.

[BV88] J.C.M. Baeten & F.W. Vaandrager, *Specification and Verification of a circuit in ACP* , Report P8803, University of Amsterdam, 1988.

[Dijk75] E.W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of programs*, in: CACM Vol. 18, pp 453-457, 1975.

[Dijk76] E.W. Dijkstra, *A discipline of Programming*, Prentice Hall, 1976.

[EM85] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specifications, Vol. I, Equations and Initial Semantics*, Springer-Verlag, 1985.

[FJKR87] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, *Formal Definition of the Design Language COLD-K*, METEOR/t7/PRLE/7, 1987.

[G88] S.J.Goldsack, *Specification of an operating system kernel: FOREST and VDM compared*, in: VDM'88 (R.Bloomfield, L.Marshall, R.Jones eds.) LNCS 328, pp. 88-100, Springer Verlag, 1988.

[Gil87] D. Gilbert, *Executable LOTOS: Using PARLOG to implement an FDT*, in: Protocol Specification, Testing, and Verification,VII, (H. Rudin & C.H. West eds.), pp 281-294, North-Holland, Amsterdam, 1987.

[GJ82] C. Ghezzi & M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, Inc., New York, 1982.

[Gla86] R.J. van Glabbeek, *Notes on the methodology of CCS and CSP*, Report CS-R8624, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[Gla87] R.J. van Glabbeek, *Bounded nondeterminism and the approximation induction principle in process algebra*, in: Proc. STACS 87 (F.J. Brandenburg, G. Vidal-Naquet & M. Wirsing, eds.), LNCS 247, pp. 336-347, Springer Verlag, 1987.

[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.

[Hend88] P.R.H. Hendriks, *ASF System User's Guide Version 1* , Annexe D13.A4 of deliverable D13 Esprit project 348 (GIPE), Third Annual Review report, Sema·Metra, Mont Rouge France, 1988.

[Hoa72] C.A.R. Hoare, *Proof of correctness of data representations*, in: Acta Informatica 1, pp 271-281, 1972.

[Hoa85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hod83] A. Hodges, *Alan Turing, The Enigma of Intelligence*, Burnett Books Limited, 1983.

[Hol83] R.C. Holt, *Concurrent Euclid, The UNIX System and Tunis*, Addison-Wesley, Reading, Massachusetts, 1983.

[INM88] INMOS Limited, *occam® 2 Reference Manual*, Prentice Hall, 1988.

[ISO86] International Organization for Standardization, *Information processing systems - Open systems interconnection - Estelle - A Formal Description Technique Based on an Extended State Transition Model*, ISO/TC 97/SC 21 N DP9074, 1986.

[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.

[Joh79] S.C. Johnson, *YACC: yet another compiler-compiler*, in: UNIX Programmer's Manual, Volume 2B, pp. 3-37, Bell Laboratories, 1979.

[LS79] M.E. Lesk & E. Schmidt, *LEX - A lexical analyzer generator*, in: UNIX Programmer's Manual, Volume 2B, pp. 39-51, Bell Laboratories, 1979.

[Mau87a] S. Mauw, *An algebraic specification of process algebra, including two examples*, Report FVI 87-06, University of Amsterdam, Amsterdam, 1987.

[Mau87b] 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.

[MV89] S. Mauw & G.J. Veltink, *An introduction to PSF_d*, in: Proc. International Joint Conference on Theory and Practice of Software Development, TAPSOFT '89, (J. Díaz, F. Orejas, eds.) LNCS 352, pp. 272-285, Springer Verlag, 1989.

[NY83] R. Nakajima & T. Yuasa, eds., *The IOTA Programming System, A modular Programming Environment*, Springer LNCS 160, 1983.

[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. Bjørner, ed.), pp. 199-225, North-Holland, 1982.

Rek87] J. Rekers, *A Parser Generator for finitely Ambiguous Context-Free Grammars*, Report CS-8712, Centre for Mathematics and Computer Science, Amsterdam, 1987.

Sha83] E.Y. Shapiro, *A subset of Concurrent Prolog and its interpreter*, Technical Report TR-003, ICOT, Tokyo, 1983.

Ued85] K. Ueda, *Guarded Horn Clauses*, Technical Report TR-103, ICOT, Tokyo, 1985.

USD80] U.S. Department of Defense, *Requirements for the Ada Programming Support Environment, STONEMAN*, 1980.

Vaa86a] F.W. Vaandrager, *Verification of two communication protocols by means of process algebra*, Report CS-R8606, Centre for Mathematics and Computer Science, Amsterdam, 1986.

Vaa86b] F.W. Vaandrager, *Process algebra semantics of POOL*, Report CS-R8629, Centre for Mathematics and Computer Science, Amsterdam, 1986.

Vra88] J.L.M. Vrancken, *The implementation of process algebra specifications in POOL-T*, Report P8807, University of Amsterdam, 1988.

Wei87] 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.)