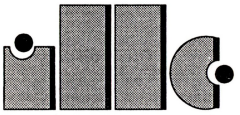
Tools for PSF

G.J. Veltink

Tools for PSF

ILLC Dissertation Series 1995-9



For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-5256090
fax: +31-20-5255101
e-mail: illc@fwi.uva.nl

TOOLS FOR PSF

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr P.W.M. de Meijer,
ten overstaan van een door het college van dekanen
ingestelde comissie in het openbaar te verdedigen
in de Aula der Universiteit
op vrijdag 9 juni 1995 te 15.30 uur

door

Gerrit Jan Veltink

geboren te Rijswijk (Z-H)

Promotor: prof. dr J.A. Bergstra

Faculteit Wiskunde en Informatica
Universiteit van Amsterdam
Kruislaan 403
1098 SJ Amsterdam

Printed and bound by CopyPrint 2000, Enschede

The work in this thesis was partially supported by ESPRIT Project no. 3006, CONCUR.
Chapter 7 was partially supported by ESPRIT Project no. 5399, COMPARE.

ISBN: 90-74795-29-3

PREFACE

Since the development of the first computers the formalisms used to program them have grown to an ever more abstract level of description. The first programmers actually had to know the bit representation of the internal *machine code* of a computer and were programming a machine only using strings of bits. It was obvious that this was not the optimal way and soon *assembler languages* were developed as a first level of abstraction.

Machine instructions are represented by an abbreviated name (mnemonic) in an assembler language. A computer program called an *assembler* is used to translate this representation into the actual machine code. As such an assembler can be regarded as the first software tool for program development.

Assembler languages are still in use nowadays for specific application areas, but the great disadvantage, besides the low level of abstraction, is that each processor (family) has its own specific assembler language based directly on the processor architecture.

The next level of abstraction was attained by the introduction of the first machine-independent programming languages such as *Fortran* and *LISP* in the late 1950's and early 1960's. Along with these languages came two types of software tools to transform programs written in such a higher-level programming language. The first tool is a *compiler* that translates a complete program into executable machine code. Another approach is taken by an *interpreter* that processes one instruction (*statement*) of the high-level programming languages at a time. After having analyzed a statement, an interpreter generates executable machine code for this statement, executes the generated machine code on the processor and fetches the next statement.

The development of programming languages has continued into the present time leading to ever more sophisticated programming languages among which many for specific application domains, eventually resulting in modern *all-purpose* languages such as *Ada* and *C++*.

Although programming languages became more powerful and offered an increasing number of structuring mechanisms it was already recognized in the early 1970's that by programming alone one would probably not be able to solve the problem of successfully implementing increasingly complex software systems. In response to this awareness a school of thought emerged, that advocated the method of giving a *specification* of a problem

before turning to its *implementation*. The first specification formalisms resulting from this idea were strongly based on the syntax of programming languages and the underlying execution model. Later, formalisms were developed based on different execution models such as *term rewriting systems* and on *propositional logic* and *first-order logic* in order to achieve a still more abstract and mathematical level of description.

The work presented in this thesis clearly belongs to the category of specification languages, however, extending it with one new concept: *concurrency*. The descriptions above implicitly assumed programming languages based on single-processor computers. The introduction of computer systems consisting of several processing units working in parallel resulted in a number of new programming concepts, as well as extra difficulties. Because parallelism is a central theme in this thesis, the nature of these difficulties will be explained in a short digression.

> A large part of the complexity in dealing with parallelism stems from the fact that even a simple parallel system may intrinsically have a large number of possible states. We will clarify this with an example. Imagine that we have to describe a crossing with four traffic lights. Each traffic light can be in one of three states, signalling *red*, *yellow* or *green*. If we consider the combination of four traffic lights working in parallel, the resulting system could theoretically be in one of 3^4 ($3 \times 3 \times 3 \times 3$) = *81* states. This number of *81* states is only valid in case all traffic lights were operating completely independently of each other. In practice, however, many of these *81* states would lead to traffic jams or even worse accidents. In a realistic specification of the crossing we aim at a much lower number of *safe* states.
>
> If we encode the states of the crossing by listing the states of the traffic lights (*r*,*y*,*g*) for the North, East, South and West branch of the crossing, in that order, we obtain the following desired five safe states: <*g*,*r*,*g*,*r*>, <*y*,*r*,*y*,*r*>, <*r*,*r*,*r*,*r*>, <*r*,*g*,*r*,*g*> and <*r*,*y*,*r*,*y*>. It is clear from the example that there is a need for a systematic way to describe such parallel processes and to be able to verify certain general properties of a system.

During the last two decades several formalisms to describe the behaviour of parallel systems have been proposed. The material in this thesis is based on one of these formalisms called the *Algebra of Communicating Processes* (*ACP*). The development of *ACP* started in 1982 at the CWI in Amsterdam. Five years later, in 1987, the design of *PSF* (*Process Specification Formalism*) initiated a project with the aim to construct a set of computer tools based on *ACP* and related formalisms. This thesis describes the set of tools that resulted from this initiative.

G.J. Veltink
Berlin, April 1994

OVERVIEW

This thesis describes the *PSF Toolkit*, a set of software tools for specifying concurrent processes based on the specification language PSF (*Process Specification Formalism*). It is divided into three main parts: *languages*, *tools* and *case studies*.

The first part of this thesis describes the languages that play a role in the implementation of the PSF Toolkit. Chapter 1 describes PSF; the main high-level language used to describe concurrent processes in this thesis. The different language constructs are introduced in an incremental way and their usage is illustrated by means of a running example that gets more complicated as more language constructs are introduced. Along with the syntax of each language construct its semantics is defined.

Although PSF is a suitable language for humans to express the behaviour of parallel processes, it is not the optimal language to be used as an interface between different computer tools. A tooling environment can strongly benefit from a layered design using different languages at different levels of abstraction. The low-level *Tool Interface Language* (TIL) used in the PSF Toolkit is introduced in Chapter 2.

Chapter 3 describes the process of translating PSF into TIL. Apart from transforming expressions from one syntax into another there is one more important transformation that is described in this chapter, namely the process of removing the modular structure from a PSF specification, also called *normalization*. PSF supports the use of *modules* to group related parts of a specification into a hierarchical design. The module concept is mainly there to be able to organize large specifications in the high-level design and as such is an aid in keeping an overview. However, on the level of computer tools such a modular structure is of no importance and one large specification, without any extra structuring mechanisms, is preferred.

The second part of the thesis focuses on the tools in the PSF Toolkit. Chapter 4 is dedicated to the simulator, a tool that simulates the behaviour of a specification. The simulator is an interactive tool, which means that, in order to study the response of a specified system in different situations, the user can influence the behaviour of the simulated system in real-time by supplying different external events. Chapter 4 starts with a description of the basic algorithmic operation of a simulator, namely calculating the *head normal form* of a process

expression. This description is an algebraic specification written in PSF itself. Then the actual implementation of the simulator along with its user interface is described.

In chapter 5 the proof assistant of the PSF Toolkit is described. The proof assistant is a tool that allows to construct proofs through a step by step transformation of process expressions. The great benefit of the tool is that it is capable of suggesting to the user all transformations possible according to the semantics of the process operators. This tool also incorporates so-called *tactics* which are special sequences of frequently used proof steps. The user interface and the implementation of the proof assistant is discussed and an example of its use is given.

Chapter 6 shortly discusses all other tools in the PSF Toolkit, including the term rewriter, the compiler driver and library manager, the initial algebra generator, the transition system generator and the equivalence tester.

The third part of this thesis deals with two case studies which have been carried out using the language PSF and the tools from the PSF Toolkit. The first case study, described in chapter 7, deals with a courseware project that evolved from a course in Software Engineering. This project aims at offering a so-called *software harness* for giving an algebraic specification of a compiler for a simple programming language. Chapter 8 gives a specification of part of the GKS (*Graphics Kernel System*) ISO Standard using an existing specification in CSP as starting point. Here it is shown that the availability of a set of tools is of great importance for checking both the syntax and correctness of a specification.

Finally chapter 9 concludes this thesis with an evaluation of the development of PSF and the PSF Toolkit. It shows the weak and strong points of the design and its implementation and gives some suggestions for improvements and future developments.

Notes:

Chapter 1 is based on chapter 2 of "*Algebraic Specification of Communication Protocols*" [MV93] and used with kind permission of Cambridge University Press.

Chapter 2 is based on: "*A Tool Interface Language for PSF*", S.Mauw & G.J. Veltink, Report P8912, Programming Research Group, University of Amsterdam. [MV89b]

Chapter 3 is a revised version of: "*From PSF to TIL*", G.J. Veltink, Report P9009, Programming Research Group, University of Amsterdam. [Vel90]

Chapter 5 is based on "*A proof assistant for PSF*" with S. Mauw [MV92], that was presented at the conference on Computer Aided Verification CAV'91 in Aalborg, Denmark.

ACKNOWLEDGEMENTS

The writing of this thesis was started while I was employed at the University of Amsterdam and finished during the time afterwards while I was employed in Germany. Many people have contributed to the research described in this thesis.

First and foremost I would like to thank *Jan Bergstra* who created a scientific atmosphere at the Programming Research Group of the University of Amsterdam with much freedom for the PSF Team, in which it was possible to experiment and learn from our mistakes. Thanks also go to *Jos Baeten* who, as supervisor of my Master's Thesis, convinced me of taking up the position of PhD Student in the first place and who was willing to review this thesis as well as to the other referees for their willingness to review this thesis and for their constructive comments: *Prof. Dr G. Engels*, *J. de Meer*, *Prof. Dr M.L. Kersten* & *Prof. Dr P. Klint*.

As co-developer of the PSF language and co-author of chapter 5 of this thesis and several other publications resulting from the PSF project, I would like to thank *Sjouke Mauw*. I would also like to thank *Pum Walters* as co-author of chapter 7.

I am very grateful to the people who collaborated in implementing the PSF Toolkit: *Bob Diertens* who implemented the simulator and proof assistant, *Casper Dik* for converting his *LISP* implementation of the term rewriter into *C* so that it could be added to the tool set and for his answers to many *UNIX*-specific questions, *Hans Mulder* for the implementation of a robust compiler driver and library manager replacing the shell scripts we used initially as well as for his sharp intuition and ability of pointing out weak spots long before anyone else was aware of them and finally *Freek Wiedijk* for adapting some of his ASF tools so that they could be used for PSF.

A piece of software depends on the feedback of its users. Therefore, thanks go to the people who served as *guinea pigs* for the use of the PSF Toolkit, thereby facing the occasional *core dump*. Nevertheless, they did not stop to contribute to the development of the tools with their valuable suggestions: *Inge Bethke*, *Jacob Brunekreef*, *Joris Hillebrand*, *Alban Ponse*, *Jos van Wamel*, *Arjan van Waveren*, *Bas van Vlijmen* and *Gert Vos*.

Several students participated in the PSF project using their research as subject for their Master's Thesis. I would like to thank for their cooperation: *Firoez Azarhoosh*, *Duncan Barrow*, *Jeroen Brouwer*, *Felix Croes*, *Henrik Jacobsson*, *Arnout Oldenburger*, *Eddie Polak* and *Wilco Koorn*.

I feel I have been lucky to work at the same site as the *Centre for Mathematics and Computers Science* (CWI). I would like to thank the following members of the scientific staff of the CWI: *Jan Willem Klop* and *Jan Heering* as well as the following people from the ASF/SDF project: *Paul Hendriks*, *Emma van der Meulen* and *Jan Rekers*.

There are many other colleagues with whom I had inspiring discussions: *Mark van den Brand*, *Nicolien Drost*, *Anton Eliëns*, *Willem Jan Fokkink*, *Rob van Glabbeek*, *Jan Friso Groote*, *Steven Klusener*, *Henri Korver*, *Karst Koymans*, *Kees Middelburg*, *Piet Rodenburg*, *Frits Vaandrager*, *Jos Vrancken*, *Peter Weijland* and *Han Zuidweg*.

Special thanks go to *David Tranah* at *Cambridge University Press* for the permission to use the material in chapter 1 which also appeared in *"Algebraic Specifications of Communication Protocols"*, and to *John Tucker* who initiated the cooperation with *Cambridge University Press*.

Although we did not cooperate in any research (in spite of the fact that we shared a room for three years), I am indebted to *Chris Verhoef*, who showed what real friendship is, even, or especially, at a distance of 700 kilometers.

A thesis is not the result of scientific research only. Therefore, I would like to express my thanks to the following people: *Madelon Drolsbach* as secretary of the *Programming Research Group* as well as all other members of the secretarial department, my parents *Bets* & *Jan Veltink* for all the support they gave me, *Silke Frevel* who showed great understanding for the time I spent in front of the screen of my *Macintosh* instead of with her, *Karin* & *Herbert Frevel* for letting me share their home during the time I was employed in Bremen and finally *Herman Maat* for moral as well as financial support.

CONTENTS

CHAPTER 1
THE LANGUAGE PSF

1.1 INTRODUCTION

In this chapter we will focus on the specification language used throughout this thesis: PSF (Process Specification Formalism). We will discuss the mathematical origins of PSF as well as its syntax and semantics. The language itself will be clarified by means of a running example, which gets more complicated as new language features are introduced.

1.2 ACP

Before we turn our attention to PSF, we will give some information on ACP (Algebra of Communicating Processes). ACP is the theoretical foundation for the process part of PSF, and deserves some explanation as such. For detailed information about the mathematical foundations of ACP we refer to the textbook [BW90]. The collected volume [Bae90] contains examples of applications of ACP, including systolic algorithms, semantics of an object oriented language and verification of protocols.

The development of ACP was started in 1982 by J.A. Bergstra and J.W. Klop, at the CWI in Amsterdam. The paper that originally introduced ACP is [BK84]. Compared with other concurrency theories like CCS [Mil80], CSP [Hoa85] and Petri Nets [Pet80,Rei85], ACP is most closely allied to CCS. The main difference between ACP and the other approaches is the way in which the semantics is treated.

Most formalisms, like CCS, CSP and Petri Nets are based on one specific model of concurrency. ACP, however, is a theory based on algebraic methods. The theory is defined by a set of axioms. The collection of possible models of this set of axioms contains most models for concurrency that have been proposed. By extending or restricting this set of axioms and by adding new operators, one can change the collection of possible models. This is a more general approach than the one that focuses on a single model. In many cases an algebraic approach towards verification has advantages over the model based approach.

1.3 The History of PSF

Several small software tools for ACP were constructed between 1982 and 1986, but the lack of a unifying framework led to many inconsistencies between those tools. In the Autumn of 1987 the PAT (Process Algebra Tools) project was started. This project aims at the construction of an integrated environment of computer tools for studying concurrent systems, especially in the setting of ACP.

The first step towards this goal was the definition of PSF, a computer readable language to specify ACP processes. Although ACP uses data types in an informal way, it was felt that PSF should incorporate data types on a more formal basis. The language definition of PSF was completed in the Spring of 1988 and the first publication on PSF was [MV89a].

The definition of data types and modularization concepts in PSF is based on the algebraic specification language ASF (Algebraic Specification Formalism) [BHK89]. The language ASF was developed at the CWI in Amsterdam.

The relation between ACP, ASF and PSF is given by the following figure. The three blocks represent the *building blocks* from which PSF has been constructed.

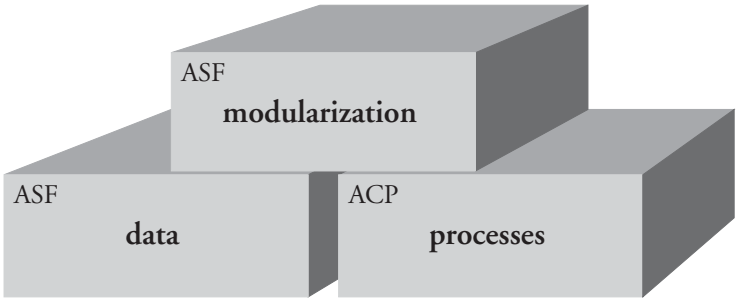


Figure 1.1 The constituent parts of PSF

After the PSF language had been defined, work on the tools started. This resulted in a compiler for PSF and several tools such as a simulator and a term rewriter.

1.4 PSF: Syntax and Semantics

In this section we will focus on the syntax and semantics of PSF. We will explain the elements of the language by specifying a simple protocol that gradually becomes more complicated as new language constructs are used. After a language construct has been introduced in an example, we will discuss its usage, general appearance and semantics.

1.4.1 Basic Operators

PSF specifications deal with the description of the activity of processes, and the interaction between different processes. The term *process* in this context must be interpreted

in a broad sense. It can range from the behaviour of a drinks dispenser and production processes in a factory to the behaviour of human beings.

The basic manifestation of activity in PSF is represented by an *atomic action*. In this thesis the terms atomic action, as well as *atom* and *action* will be used. Each atomic action has a name or *label* and whenever an atomic action with name a is active, we say that a happens or is executed.

Processes in PSF are defined in terms of *process expressions*. The atomic action is the basic process expression. Complex process behaviour in PSF is expressed by combining process expressions in different ways using additional *operators*. In PSF there are for instance operators that tell whether two process expressions are to be executed in succession, simultaneously or that a choice has to be made between either one of them.

1.4.1.1 ACTION RELATIONS

To attach *semantics* or meaning to the operators, we describe their behaviour by means of a mathematical notation called *action relations*. This technique was first used in [Plo82] to define an operational semantics for CSP.

For each atomic action a we define a binary relation and a unary relation on process expressions. The notation of the two relations is given below. The dots represent process expressions.

$$\bullet \xrightarrow{a} \bullet \qquad \bullet \xrightarrow{a} \surd$$

Table 1.1 Action relations

When x and y are variables for process expressions, the notation $x \xrightarrow{a} y$ expresses the fact that x can evolve into y by executing the atomic action a. The notation for the unary relation $x \xrightarrow{a} \surd$ is used to express that x, after having executed atomic action a, has reached a state in which it terminates successfully. The $\surd$ symbol (tick) indicates successful termination of a process.

The simplest action relation states that a process expression consisting of a single atomic action a, can terminate successfully by executing this atomic action a. This action rule is given by:

$$a \xrightarrow{a} \surd$$

Table 1.2 Action relation for atomic action

1.4.1.2 THE EXAMPLE

The running example that we will use, is an abstraction of a specification of a communication protocol. The example describes, like most protocol specifications, a pair of a sender and a receiver that are connected to each other by a channel. The goal is to transmit data from the sender to the receiver via the channel.

The graphical representation of the example is given below. The three different processes are represented by boxes. The dots in the boxes represent the atomic actions that a

process is able to perform. The interaction between the different components is not modelled.

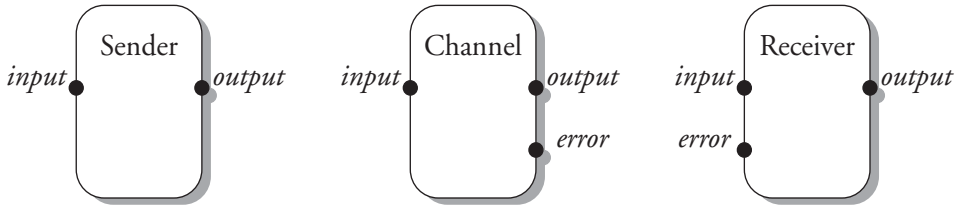


Figure 1.2 Graphical representation of the components of the running example

1.4.1.3 IDENTIFIER NAMES, LEXICAL CONVENTIONS

Entities in a PSF specification are referred to by a name. These names or *identifiers* must meet some lexical requirements. The *identifier character set* of PSF consists of all lower case characters (*a, b, …, z*), all upper case characters (*A, B, …, Z*), all digits (*0, 1, …, 9*) and the special symbols *single quote* (') and *hyphen* (-). A valid identifier in PSF is a string starting with a lower case or upper case character, followed by zero or more characters from the identifier character set.

In this thesis we will print all keywords in PSF specifications in bold face. This is not part of the lexical conventions of PSF, but only serves to increase the legibility of the specifications.

1.4.1.4 COMMENTS

Comments in PSF are introduced by two consecutive hyphens (--). Comments are closed by another pair of hyphens or by the *newline* character, whichever one comes first. Within the text of the comment, characters from the complete character set of the host computer can be used, with the exception of the above-mentioned newline and pair of hyphens.

1.4.1.5 ATOMS & PROCESSES

The first PSF specification, representing the sender from Figure 1.2, is given below:

```
process module Sender
begin

  atoms
    input, output

  processes
    Sender

  definitions
    Sender = input . output . Sender

end Sender
```

The specification is given in the form of a *module*. The use of modules will be explained in more detail in the section on modularization. The *module* construct in this example serves three purposes:

- it groups a set of related entities, using a *begin-end* pair;
- it gives the type of the module: *process* (as opposed to a *data* module);
- it attaches a name to the module: *Sender*.

Within the module we see different sections. The *atoms* section defines a set of atomic actions by declaring their names. The *processes* section declares a set of process names. Finally, the processes are defined in the *definitions* section by relating a process name to a process expression using the '=' construct. In the example the behaviour of *Sender* is defined in terms of the two actions *input* and *output* and the process *Sender* itself. We should mention two technical points here:

- if there is more than one definition for a process in the *definitions* section, the different process expressions will be interpreted as *alternatives*. The concept of *alternatives* in PSF will be explained in one of the following sections.
- if a process name is declared but has not been defined, it is defined implicitly to be *deadlock*. The constant process expression *deadlock* will be described in the following section.

Although the language permits the usage of any valid identifier name for the different entities, we will adhere to some standard naming convention. This will help to distinguish between the different identifiers in a specification.

- a module name starts with an upper case character.
  example: *Sender*.
- an action name consists of lower case characters only.
  example: *input*.
- a process name starts with an upper case character.
  example: *Sender*.

1.4.1.6 DEADLOCK

Although *deadlock* will never literally appear in a PSF specification, we have to explain its behaviour. We can think of deadlock as a process expression describing a state, from which no progression can be made. In other words: deadlock is an *unsuccessful termination*. Because there are no actions that can evolve from a deadlock, we will not associate any action relation with it.

1.4.1.7 SEQUENTIAL COMPOSITION

The process expression defining the behaviour of the process *Sender*, consists of atomic actions: *input* and *output*, the process *Sender* itself and the '·'-operator. This operator is called the *sequential composition*, and expresses the order in which events have to occur. The semantics for the sequential composition is given by the following action relations. The expressions above the line are conditions that must be met to conclude the expressions below the line.

$$\frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \xrightarrow{a} \surd}{x \cdot y \xrightarrow{a} y}$$

Table 1.3 Action relations for sequential composition

The first rule says that whenever x can execute an a thereby evolving into x', $x \cdot y$ can perform this same a, and evolve into $x' \cdot y$. The second rule states that whenever x can terminate successfully after having executed an a, the process $x \cdot y$ can evolve into y after an a.

In the example of the sender, the process expression $input \cdot output \cdot Sender$ executes the action *input*, followed by *output*. The remaining part of the process expression, after these actions have been executed, is *Sender*.

1.4.1.8 RECURSION

Process expressions can contain process names, as was shown in the previous section. The occurrence of such a process name is in fact an abbreviation for the process expression related to the process name as defined in the definitions section.

If the name of a process appears in its own definition, directly or indirectly through a chain of expansions of other process names, we call this phenomenon *recursion*. Using recursion it is possible to specify processes with an infinite behaviour, by means of a finite set of finite definitions. The action relations describing recursion are given below. The X denotes the process name and x the process expression it represents.

$$\frac{X = x \ \wedge \ x \xrightarrow{a} x'}{X \xrightarrow{a} x'} \qquad \frac{X = x \ \wedge \ x \xrightarrow{a} \surd}{X \xrightarrow{a} \surd}$$

Table 1.4 Action relations for recursion

In the running example the sender, which is an infinite process, is defined by means of recursion.

1.4.1.9 ALTERNATIVE COMPOSITION

In the next specification we describe the channel from Figure 1.2. The channel we are going to specify, allows for data to travel in one direction and has the possibility to lose the data it is transporting. The behaviour is described as follows: first the channel reads a datum at its input port, then it decides whether to pass it on to its output port correctly or incorrectly. Finally it returns to its initial state, waiting for the next datum to arrive at the input port.

```
process module Channel
begin

  atoms
    input, output, error

  processes
    Channel, Choice
```

```
    definitions
      Channel = input . Choice . Channel
      Choice = output + error

end Channel
```

The choice determining whether or not a datum is transported correctly is modelled by the process *Choice* through the use of the '+'-operator, the *alternative composition*. The process expression $x + y$ denotes a process that chooses exactly one of the possible initial actions of x and y. However, a possible deadlock (initial) action is not chosen if there are alternatives. The semantics for the alternative composition is given by the following action rules:

$$\frac{x \xrightarrow{a} x'}{x+y \xrightarrow{a} x'} \qquad \frac{x \xrightarrow{a} \surd}{x+y \xrightarrow{a} \surd} \qquad \frac{y \xrightarrow{a} y'}{x+y \xrightarrow{a} y'} \qquad \frac{y \xrightarrow{a} \surd}{x+y \xrightarrow{a} \surd}$$

Table 1.5 Action relations for alternative composition

To complete the specification of the example we give the specification of the receiver. Its behaviour is described as follows. It has to anticipate the incorrect transmission of the datum through the channel. Therefore it can either perform an *input* or an *error* action in its initial state. If the *input* action occurs, a corresponding *output* action is performed and the process returns to its initial state. However, if the *error* action occurs, the process returns to the initial state immediately, without performing any further actions.

```
process module Receiver
begin

  atoms
    input, output, error

  processes
    Receiver

  definitions
    Receiver = ((input . output) + error) . Receiver

end Receiver
```

The example shows that parentheses can be used, in the process definition, to group expressions. The sequential composition binds stronger than the alternative composition, so in the example the inner pair of parentheses could have been left out.

1.4.1.10 INTEGRATING COMPONENTS

Now that we have given specifications for the different components of the system we want to integrate them. In this section we will combine them into one module. In the section on modularization we will discuss a different approach to integration.

The first problem that we come across when we try to put the components in one module, is a naming problem. Each component in the original specification contained an *input* and an *output* action. If we would simply take the union of the different sections, the name *input* would become ambiguous, because it would refer to three different atomic actions. This

phenomenon is called a *name clash*. To overcome this problem we have to assign new names to the different atomic actions.

The specification of the complete system is given below. It contains quite some new language constructs, which will be explained in detail in the sections to come.

```
process module System
begin

  atoms
    sender-input, channel-input, receiver-input,
    sender-output, channel-output, receiver-output,
    channel-error, receiver-error,
    sender-to-channel, channel-to-receiver, error-transmission

  processes
    System, System', Sender, Channel, Receiver

  sets
    of atoms
      H = { sender-output, channel-input, channel-output,
            receiver-input, channel-error, receiver-error }
      I = { sender-to-channel, channel-to-receiver, error-transmission }

  communications
    sender-output | channel-input = sender-to-channel
    channel-output | receiver-input = channel-to-receiver
    channel-error | receiver-error = error-transmission

  definitions
    Sender = sender-input . sender-output . Sender
    Channel = channel-input .
                (skip .  channel-output + skip . channel-error) .
               Channel
    Receiver = ((receiver-input . receiver-output) + receiver-error) .
                Receiver
    System = encaps(H, Sender || Channel || Receiver)
    System' = hide(I, System)

end System
```

1.4.1.11 SETS

The first new section that appears in the specification of the complete system is the *sets* section. In this case two *sets of atoms* are defined. Sets are merely a notational convenience and are used as parameters in certain *process operators*.

We can assign a name to a set, so that we do not have to repeat the contents of the set every time we want to refer to it. In the example, H and I are defined by an *enumeration* of the atomic actions that are part of the set. Moreover, there are three set operators: *union*, *intersection* and *difference*. Because we need to have a representation for these operators in standard ASCII format, we are not able to use to standard notation $\cup$ and $\cap$ for union and intersection. The PSF syntax for these operators is:

- enumeration : $\{ s_1, s_2, \ldots, s_n \}$ s_i is a set element
- union : $S + T$ S and T are sets
- intersection : $S \,.\, T$
- difference : $S \setminus T$

The naming convention for sets is as follows:

- a set name starts with an upper case character.
  example: *H*.

It is also possible to define sets that are not sets of atoms. We will see examples of that at a later stage.

1.4.1.12 COMMUNICATION

The other new section that appears in the example is the *communications* section. In this section the possibilities of communication between the different atomic actions are described.

In PSF, communication is defined as the simultaneous occurrence of two atomic actions. As a result of the communication these two atomic actions are transformed into one new atomic action. The communication function, that tells which actions are allowed to communicate and what their result will be, is defined in the communications section. The communication definition $a \mid b = c$, states that the atomic actions a and b can communicate and that the resulting action will be c. We should make two technical remarks with respect to the definition of the communication function:

- for any pair of actions at most one resulting action can be defined.
- there are no communications possible, other than the ones that are explicitly defined.

1.4.1.13 PARALLEL COMPOSITION

The *definitions* section of the example introduces a new operator: the *parallel composition*. The expression $x \parallel y$ states that the processes x and y are executed in parallel. PSF is based on a so-called *interleaving* semantics [BW90] for parallel composition. This means that either the left operand or the right operand performs an atomic action or that the initial actions from both operands communicate with each other. The action rules for parallel composition are:

$$\frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad \frac{x \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} y} \qquad \frac{x \xrightarrow{a} x' \wedge y \xrightarrow{b} y' \wedge a \mid b = c}{x \parallel y \xrightarrow{c} x' \parallel y'} \qquad \frac{x \xrightarrow{a} \surd \wedge y \xrightarrow{b} y' \wedge a \mid b = c}{x \parallel y \xrightarrow{c} y'}$$

$$\frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \qquad \frac{y \xrightarrow{a} \surd}{x \parallel y \xrightarrow{a} x} \qquad \frac{x \xrightarrow{a} x' \wedge y \xrightarrow{b} \surd \wedge a \mid b = c}{x \parallel y \xrightarrow{c} x'} \qquad \frac{x \xrightarrow{a} \surd \wedge y \xrightarrow{b} \surd \wedge a \mid b = c}{x \parallel y \xrightarrow{c} \surd}$$

Table 1.6 Action relations for parallel composition

The parallel composition binds stronger than the alternative composition, but less strongly than the sequential composition.

1.4.1.14 ENCAPSULATION

The *encapsulation operator*, in a PSF specification written for example as: *encaps(H,x)*, is a specific instance of the general class of *renaming* operators. Renaming operators rename atomic actions into process expressions. The encapsulation operator takes as arguments a set

of atoms H, and a process expression x. All atomic actions from H, that appear in the process expression are renamed into *deadlock*. This is also true for actions that are the result of some communication that appears inside the process expression. The action relations are as follows:

$$\frac{x \xrightarrow{a} x' \wedge a \notin H}{\mathbf{encaps}(H,x) \xrightarrow{a} \mathbf{encaps}(H,x')} \qquad \frac{x \xrightarrow{a} \surd \wedge a \notin H}{\mathbf{encaps}(H,x) \xrightarrow{a} \surd}$$

Table 1.7 Action relations for the encapsulation operator

There are no action relations for the case $a \in H$. The complete *encapsulation expression* would be renamed into *deadlock*, from which it is not possible to do any steps.

The main usage of the encapsulation operator is to guide the process of communication between two processes. In the previous section we have seen that the parallel composition of x and y is defined as the alternative composition of three other process expressions. In most cases we want just one of the three alternatives to appear, namely the one in which an atomic action from x and an atomic action from y communicate. By adding the atomic actions that appear in x and y to the *encapsulation set*, we obstruct them from being performed as such, leaving only the possibility in which x and y have to communicate.

1.4.1.15 THE INTERNAL STEP

The specification of the channel uses a new constant process expression: *skip*. This *skip* is an internal action of the system. We will encounter its main usage in the following section. The action rule for *skip* is:

$$\mathbf{skip} \xrightarrow{\text{skip}} \surd$$

Table 1.8 Action relation for the internal step

In the running example *skip* is used to make sure that the choice between successful and unsuccessful transmission of the data is made non-deterministically. In other words, the choice cannot be influenced by the outside world.

If we would leave out the *skip* expressions, the receiver could influence the choice by offering only one of the possible partners in communication, thereby forcing a choice. Now the choice is made internally between two indistinguishable *skips*.

We should mention a technical point here. The expression *(skip · input) + (skip · error)* is not equivalent to *skip · (input + error)* in the semantics of PSF. In other words: the left-distributivity of · over + does not hold, although the right-distributivity does.

The difference stems from the fact that the *moment of choice* in both expressions is not the same. Consider the following two specifications of the "unevitable coffee machine"† :

```
Vending-Machine = coin . (coffee + tea)
Funny-Vending-Machine = (coin . coffee) + (coin . tea)
```

† See [Mul90] for more elaborate examples of coffee machines

The *Vending-Machine* accepts a coin and then offers to choose between coffee and tea. The *Funny-Vending-Machine* also accepts a coin, but in accepting the coin it chooses internally between the two branches *coin · coffee* and *coin · tea*. This implies that we, the outside world, do not have any possibility to influence this choice and never can be sure whether we get coffee or tea for our coin.

1.4.1.16 ABSTRACTION

The *abstraction operator*, in the PSF specification written as *hide(I,x)*, is closely related to the encapsulation operator. The abstraction operator renames atomic actions, contained in its set argument *I*, into *skip*. The action rules are as follows:

$$\frac{x \xrightarrow{a} x' \wedge a \in I}{\mathbf{hide}(I,x) \xrightarrow{skip} \mathbf{hide}(I,x')} \qquad \frac{x \xrightarrow{a} \surd \wedge a \in I}{\mathbf{hide}(I,x) \xrightarrow{skip} \surd}$$

$$\frac{x \xrightarrow{a} x' \wedge a \notin I}{\mathbf{hide}(I,x) \xrightarrow{a} \mathbf{hide}(I,x')} \qquad \frac{x \xrightarrow{a} \surd \wedge a \notin I}{\mathbf{hide}(I,x) \xrightarrow{a} \surd}$$

Table 1.9 Action relations for the abstraction operator

The main usage of the abstraction operator is also connected with the process of communication. In a large number of specifications we are not interested in the result of all the communication actions. Many of the communication actions are just used internally to synchronize two or more processes. When investigating the behaviour of a process, we are mostly interested in the *external* behaviour. This means that we want to abstract from all *internal* activity. We can achieve this by renaming all atomic actions that represent details we are not interested in, into *skip*.

As suggested before, the *abstraction operator* and *encapsulation operator* are mostly used in conjunction. A typical example is:

hide(I, encaps(H, …))

In the running example we have also used this construct to specify the process *System'*. We abstract from all internal communications between the sender, the channel and the receiver (*sender-to-channel, channel-to-receiver, error-transmission*). These actions form the set *I*. The set *H* consists of all atomic actions that take part in the communications that result in the actions from set *I*. By encapsulating the actions from *H* we force communication between them. The result is that in *System'* the only observable behaviour consists of the actions *sender-input* and *receiver-output* and of the constant process expression *skip*.

1.4.1.17 SUMMARY OF THE PROCESS PART OF PSF

In this section we will give a graphical representation of the relation between the concepts we have introduced so far. The ellipses represent the different domains we have explained. The arrows suggest the different functions that take elements from a domain (the dots), and produce an entity in another domain (the arrow head).

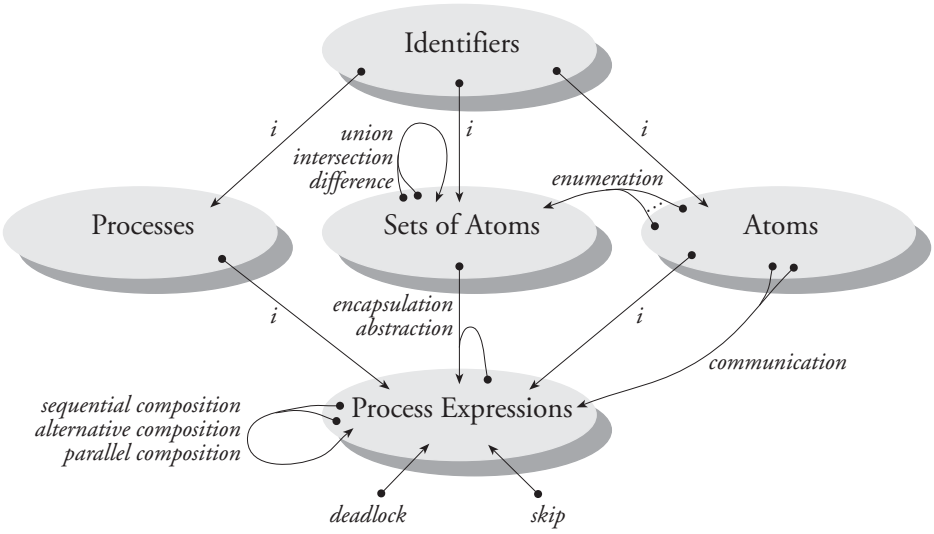


Figure 1.3 Graphical representation of relations between PSF concepts

The function *i* represents the *injection* function. The injection function is an *implicit*, *invisible*, function application. To clarify this notion we will give an example.

The string *a* is an identifier. Moreover it can be the name of a process, a set or an atomic action. Whenever it is a name for a process or atomic action, it is also a process expression representing the corresponding process or atomic action. Its type is derived from the context in which it is used.

1.4.2 MODULARIZATION

In the previous section we have integrated different components by combining them into one module. This method is rather cumbersome and becomes more and more complicated as the size of the specification grows. PSF supports the use of several modularization techniques, which will be discussed in the sequel.

1.4.2.1 MODULES

In the previous specifications we already have used the *module* concept. Each specification consisted of exactly one module. A general PSF specification consists of a series of modules. Modules can depend on objects defined in other modules, so there exists a certain relation between them.

1.4.2.2 EXPORTS

The first step towards modular design is to divide all the entities in a module into two categories. One category for the objects that are visible to the world outside a module, the *exported objects*, and one for those objects that are used only locally, the *hidden objects*.

In PSF the exported objects are surrounded by a *begin-end* pair that follows the *exports* keyword. All other objects that are defined are hidden. In the two examples below all objects are exported. These examples redefine the *Sender* and *Receiver* processes we have defined earlier.

```
process module Sender
begin

  exports
    begin
      atoms
        input, output
      processes
        Sender
    end

  definitions
    Sender = input . output . Sender

end Sender

process module Receiver
begin

  exports
    begin
      atoms
        input, output, error
      processes
        Receiver
    end

  definitions
    Receiver = ((input . output) + error) . Receiver

end Receiver
```

1.4.2.3 HIDDEN OBJECTS

To show the use of *hidden* objects we give the specification of a module in which the exported process is defined in terms of an *auxiliary* process. Because of its very specific nature, this auxiliary process is intended to be local to the module. This feature of shielding functions and data from the outside world is also known as *information hiding* or *data encapsulation*. The term *hidden* in the context of modularization should not be confused with *hide*, the representation of the abstraction operator in PSF. The following example redefines the *Channel* process.

```
process module Channel
begin

  exports
    begin
      atoms
        input, output, error
      processes
        Channel
    end
```

```
  processes
    Choice

  definitions
    Channel = input . Choice . Channel
    Choice = (skip . output) + (skip . error)

end Channel
```

The process *Choice* is the local process, used in the definition of *Channel*. It is not meant to be known to the outside world and therefore declared to be hidden. We recall that all entities that are not explicitly named in the *exports* section are hidden.

1.4.2.4 IMPORTING MODULES & RENAMING OBJECTS

Now we will show how the three modules we have defined in the previous sections can be combined by means of *imports,* into one module called *System*.

```
process module System
begin

  exports
    begin
      atoms
        sender-to-channel, channel-to-receiver, error-transmission
      processes
        System, System'
    end

  imports
    Sender {
      renamed by [
        input -> sender-input,
        output -> sender-output
      ]
    },
    Channel {
      renamed by [
        input -> channel-input,
        output -> channel-output,
        error -> channel-error
      ]
    },
    Receiver {
      renamed by [
        input -> receiver-input,
        output -> receiver-output,
        error -> receiver-error
      ]
    }

  sets
    of atoms
      H = { sender-output, channel-input, channel-output,
            receiver-input, channel-error, receiver-error }
      I = { sender-to-channel, channel-to-receiver, error-transmission }
```

```
  communications
    sender-output | channel-input = sender-to-channel
    channel-output | receiver-input = channel-to-receiver
    channel-error | receiver-error = error-transmission

  definitions
    System = encaps(H, Sender || Channel || Receiver)
    System' = hide(I, System)

end System
```

In the example above, the exported objects from the module *Sender* are made available to *System* by importing *Sender*. The imported modules are given as a list of modules preceded by the *imports* keyword.

In the example we see that every module name is followed by a list of *renamings*. The renamings first specify the name of the object in the module of origin, and then the name in the current module. If an object is not renamed in the renaming list, it keeps its old name.

Although renaming is optional, in the example we have to rename the atomic actions coming from the imported modules, because otherwise we would run into a name clash, as was explained earlier.

All objects that are imported into a module, are implicitly imported into its export section. This means for example, that any module that imports module *System* will automatically be able to use the action *sender-input* from module *Sender*.

1.4.3 Data Modules & Parameterization

Until now we have given examples in which we have been abstracting from the identity of messages sent from the sender to the receiver. We have been describing the behaviour of the system for an abstract message. If we want to model our example on a higher level of detail, we have to be able to add data parameters to the atomic actions and processes. In the following sections we will show how data is modelled in PSF, and how it is incorporated into the processes. Moreover, we will show how a module can be parameterized with elements from the data domain.

1.4.3.1 Sorts & Functions

The specification of data types in PSF is given using a so-called *equational specification* [EM85] and is based on the algebraic specification language ASF [BHK89]. The entities that are used in such specifications are *sorts* and *functions*. In the example below we give a specification of the *booleans*.

```
data module Booleans
begin

  exports
    begin
      sorts
        BOOLEAN
      functions
        true : -> BOOLEAN
        false : -> BOOLEAN
    end

end Booleans
```

As opposed to the process modules, a data module is preceded by the keyword: *data*. In the example we see that there are two new sections. The *sorts* section introduces sorts or data domains, and the functions section declares functions that operate on elements from these data domains.

The example defines one sort: *BOOLEAN*, and two functions *true* and *false*. The naming convention follows from the example:

- a sort name consists of upper case characters only.
  example: *BOOLEAN*.
- a function name consists of lower case characters only.
  example: *true*.

A function declaration has the following general appearance:

function-name : *input type* -> *output type*

The clause *input type* is a list of sorts separated by '#'-symbols and *output type* is a sort. The input type specifies the types of the arguments of the function, and the output type specifies the type of the result. The two functions in the example have no input type and are so-called *constant functions*.

1.4.3.2 EQUATIONS

The behaviour of functions is given by means of equations. In an equation we actually specify that two terms are equivalent. In the implementation however, an equation is interpreted as a rule from a *term rewriting system*, in which the left-hand side is rewritten to the right-hand side. In the example below we will give a specification of a module containing some messages.

```
data module Messages
begin

  exports
    begin
      sorts
       MESSAGE
      functions
       message-error : -> MESSAGE
       message-1 : -> MESSAGE
       message-2 : -> MESSAGE
       message-3 : -> MESSAGE
       valid : MESSAGE -> BOOLEAN
    end

  imports
    Booleans

  equations
  [01] valid(message-error) = false
  [02] valid(message-1) = true
  [03] valid(message-2) = true
  [04] valid(message-3) = true

end Messages
```

We have introduced the sort *MESSAGE*, that contains four messages. One of these items represents a message that has been garbled: *message-error*. To be able to distinguish between valid and invalid messages we have defined a function *valid*, which accepts a message as input and which yields the boolean value *true* for an ordinary message, and *false* for a garbled message. The usage of the function *valid* will be illustrated in one of the following sections.

1.4.3.3 INITIAL ALGEBRA SEMANTICS

In the previous sections we have been assigning semantics to all language constructs we introduced using *action relations*. The data type specifications in PSF are interpreted using *initial algebra* semantics [EM85]. We will only sketch the principles of the initial algebra of a specification here. For a more detailed explanation we refer to the literature.

Using the functions introduced in a data type specification it is possible to construct *terms*. Two examples of simple *terms* of type *BOOLEAN*, from module *Booleans*, are *true* and *false*.

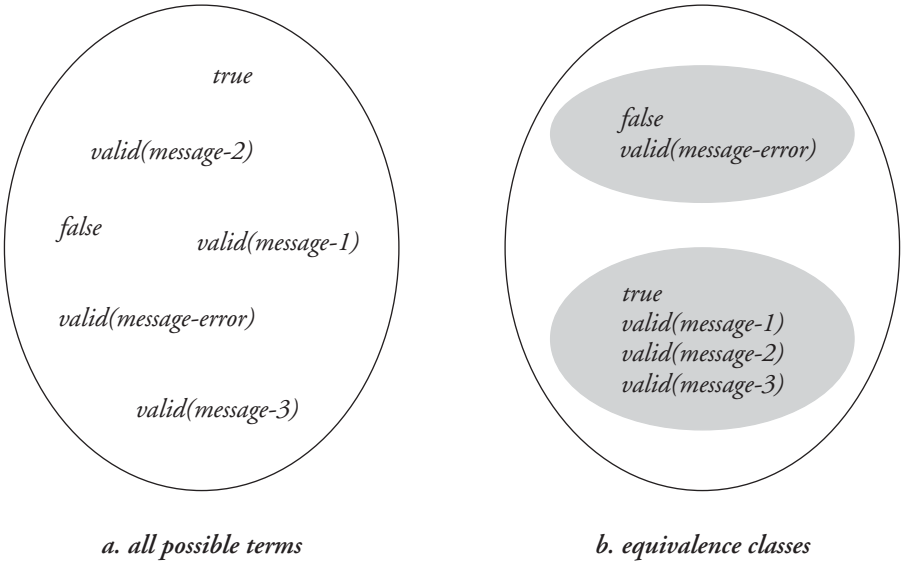


Figure 1.4 Determining the initial algebra

Module *Messages* introduces four constant functions of type *MESSAGE* and a function *valid* with *MESSAGE* as input type and *BOOLEAN* as output type. Using *valid* it is possible to create four new terms of type *BOOLEAN*: *valid*(*message-error*), *valid*(*message-1*), *valid*(*message-2*) and *valid*(*message-3*). This implies that there are six different terms of type *BOOLEAN* in module *Messages*, the four created using *valid*, plus the two already introduced in *Booleans*.

Not all these terms refer to different entities. One of the equations in *Messages* states that *valid*(*message-1*) is equal to *true*. We say that *valid*(*message-1*) and *true* refer to the same element in the *initial algebra* of *BOOLEAN*.

We conclude with the description of an informal method to determine the initial algebra of a sort. The first step is to generate all possible terms of a given sort. The result is a possibly infinite set of data terms. In the second step all terms that are equal, according to the given equations, are grouped together. The result is that the set of terms is divided into so-called *equivalence classes*. Each equivalence class corresponds to exactly one element of the initial algebra. One of the terms in the equivalence class is chosen to be the *representant* of the given equivalence class. Figure 1.4 shows the above-mentioned procedure for the sort *BOOLEAN*.

1.4.3.4 Parameterization of Actions and Processes

The data introduced in the previous section, is integrated into the process domain, by allowing processes and actions to be parameterized with tuples of data elements. To put it in a different way, processes and actions can have data items as arguments.

To show the use of these data parameters, and not have to introduce too many new concepts at once, we shortly leave our running example and turn to a specification of a memory cell that can contain one *boolean* value. First we will give a graphical representation of this memory cell.

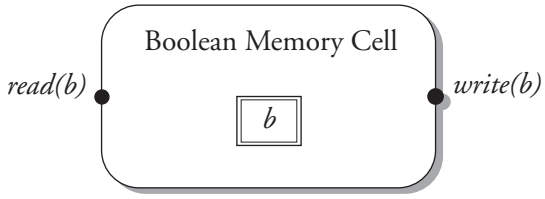


Figure 1.5 Boolean Memory Cell

The behaviour of the memory cell is as follows. In the initial state it is only capable of reading a boolean value performing the action *read*. After having read a value it can either input a new value or it can show the value it contains by performing the action *write*. The actions will be parameterized with a boolean value.

Keep in mind that this is a specification of a memory cell from the point of perspective of the cell itself. So *read* does *not* mean reading the memory cell but assigning a value to the memory cell.

```
process module Boolean-Cell
begin

  imports
    Booleans

  atoms
    read, write : BOOLEAN

  processes
    Cell
    Cell : BOOLEAN

  variables
    b : -> BOOLEAN
```

```
  definitions
    Cell    = sum(v in BOOLEAN, read(v) . Cell(v))
    Cell(b) = write(b) . Cell(b) + Cell

end Boolean-Cell
```

Arguments of an action or process are written between parentheses, as shown in the example. The actions *read* and *write* are declared to have a boolean argument. There are two instances of the process *Cell*, one with a boolean parameter, and one with no parameters. This is an example of a phenomenon called *overloading*.

1.4.3.5 OVERLOADING OF IDENTIFIERS

The name *Cell* is overloaded, because it can refer to two different processes. PSF allows overloading of the names of entities as long as the entities can be distinguished by the types of their parameters. Entities that cannot contain parameters, like sorts for instance, may not be overloaded.

1.4.3.6 VARIABLES

The example introduces one new section: the *variables* section. In this section we define the type of the variables that are used in the process definitions. In this example the process *Cell(b)* uses its parameter *b*, a boolean variable, to remember its current value. Variables cannot be exported and are therefore always local to the module in which they are defined.

1.4.3.7 GENERALIZED ALTERNATIVE AND PARALLEL COMPOSITION

The final point of interest in the example of the memory cell, is the usage of the generalized alternative composition: s*um(v in BOOLEAN, …)*. The general appearance of the generalized alternative composition is: *sum(x in S, P)*. The *S* can be a *sort* or a *data* set. The *x* is called a *placeholder* and represents an element from the data domain of *S*. The expression *sum(x in S, P)* is an abbreviation of an alternative composition containing, for every possible *valuation* of *x*, a copy of *P* in which *x* is replaced by the valuation of *x*:

$$P[x/s_1] + \ldots + P[x/s_n]$$

As an example of this substitution we give the evaluation of the definition of *Cell* from the example:

$$read(true)\ .\ Cell(true) + read(false)\ .\ Cell(false)$$

In the same way we define the generalized parallel composition, which is denoted by *merge(x in S, P)*.

The specification of the memory cell represents a system that can be in any of three *states*: *Cell*, *Cell(false)* and *Cell(true)*. Figure 1.6 shows the states and the possible transitions between them.

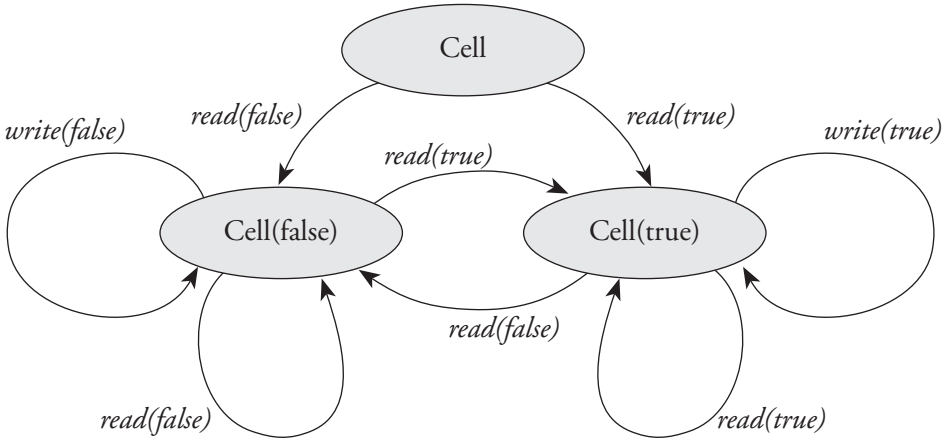


Figure 1.6 States and transitions of the memory cell

1.4.3.8 PARAMETERIZATION OF A MODULE

In the previous sections we have shown how a data element can be a parameter of a process or an atomic action. In this section we will focus on parameterization on a different level, namely on the level of modules.

In the approach we took for the specification of the memory cell, we specified a cell that explicitly deals with booleans. If we would like to give a specification of a cell that remembers natural numbers, we would have to construct a complete new specification. This is of course inconvenient, and therefore PSF incorporates a mechanism that enables us to give a specification of a process that operates on a certain data type, although that type itself is not yet fully specified.

The *Sender* process is a good candidate for parameterization, because in essence it is capable of sending all kinds of objects. Therefore, we can make the data type a parameter of the module.

```
process module Sender
begin

  parameters
    Sender-Parameter
    begin
      sorts
        DATA
    end Sender-Parameter

  exports
    begin
      atoms
        input : DATA
        output : DATA
      processes
        Sender
    end
```

```
  definitions
    Sender = sum(d in DATA, input(d) . output(d)) . Sender

end Sender
```

In the example above we see that parameters are introduced in the *parameters* section. The entries in the *parameters* section are *named parameter blocks*. The name of the parameter block in the example is *Sender-Parameter*. Each block can contain *sorts*, *functions*, *sets*, *atoms* and *processes*. The entities that are introduced in the parameters section, are treated in the rest of the module as if they were normal entities.

Not all processes can be parameterized in such a simple way as described above. The *Sender* does not *depend* on the data it is transporting. On the contrary, a process that would sort items *does* depend on the data it is sorting, because it has to know the order of two data items. In such cases, the module that defines the sorting process still can be parameterized, but the ordering function must also be part of the parameter.

The following specification of the *Channel* also shows such a dependency. The channel can transport a datum correctly or the datum can get garbled. In the latter case the channel has to output a special *error element* of type *DATA*. This error element is therefore part of the parameter of *Channel*.

```
process module Channel
begin

  parameters
    Channel-Parameter
    begin
      sorts
        DATA
      functions
        error : -> DATA
    end Channel-Parameter

  exports
    begin
      atoms
        input : DATA
        output : DATA
      processes
        Channel
  end

  processes
    Choice : DATA

  variables
    d : -> DATA

  definitions
    Channel = sum(d in DATA, input(d) . Choice(d)) . Channel
    Choice(d) = (skip . output(d)) + (skip . output(error))

end Channel
```

1.4.3.9 CONDITIONAL EXPRESSIONS

The specification of the *receiver* with data parameters, requires a new language construct, the *conditional expression*. The conditional expression was not part of the initial language definition of PSF and has been added at a later stage. It was first introduced in a paper describing the language PSF/C [BBMV91]. PSF/C is a dialect of PSF that uses COLD [FJ93] instead of ASF to represent data types.

The conditional expression is used to make choices, based on data terms, thereby influencing the behaviour of the process. In the example below, the *conditional expression* is used to check the validity of the incoming data and to respond to it in different ways.

```
process module Receiver
begin

  parameters
    Receiver-Parameter
    begin
      sorts
        DATA
      functions
        valid-data : DATA -> BOOLEAN
    end Receiver-Parameter

  exports
    begin
      atoms
        input : DATA
        output : DATA
      processes
        Receiver
    end

  imports
    Booleans

  processes
    Validate : DATA

  variables
    d : -> DATA

  definitions
    Receiver = sum(d in DATA, input(d) . Validate(d))
    Validate(d) = [valid-data(d) = true] -> output(d) . Receiver +
                  [valid-data(d) = false] -> Receiver
end Receiver
```

Each process expression P can be prefixed by a conditional expression as in $[s = t] \rightarrow P$. The expression $[s = t]$ is also called a *guard*. If s and t are equal, according to the initial algebra, we say that the guard is $true$, if they are not equal the guard is *false*. If the guard of $[s = t] \rightarrow P$ is true, the conditional expression evaluates to process P. If the guard is false, the conditional expression evaluates to *deadlock*. The semantics is given below.

$$\frac{x \xrightarrow{a} x' \wedge \ s = t}{([s = t] \rightarrow x) \xrightarrow{a} x'} \qquad \frac{x \xrightarrow{a} \surd \wedge \ s = t}{([s = t] \rightarrow x) \xrightarrow{a} \surd}$$

Table 1.10 Action relations for the conditional expression

1.4.3.10 BINDING OF PARAMETERS

After having carried out such a conscientious preparation, we are finally able to create a completed specification of the interaction between the sender, the channel and the receiver. We achieve our final specification of the running example by binding all parameters, the *formal* entities, in the specification to *actual* entities. In the following example the sort *DATA*, which was the parameter representing the type of the objects that are handled by the protocol, is bound to the sort *MESSAGE* of the module *Messages*.

```
process module System
begin
  exports
    begin
      atoms
        sender-to-channel : MESSAGE
        channel-to-receiver : MESSAGE
      processes
        System, System'
    end
  imports
    Sender {
      Sender-Parameter bound by [
        DATA -> MESSAGE
      ] to Messages
      renamed by [
        input -> sender-input,
        output -> sender-output
      ]
    },
    Channel {
      Channel-Parameter bound by [
        DATA -> MESSAGE,
        error -> message-error
      ] to Messages
      renamed by [
        input -> channel-input,
        output -> channel-output
      ]
    },
    Receiver {
      Receiver-Parameter bound by [
        DATA -> MESSAGE,
        valid-data -> valid
      ] to Messages
      renamed by [
        input -> receiver-input,
        output -> receiver-output
      ]
    }
```

```
  sets
    of atoms
      H = { sender-output(m), channel-input(m), channel-output(m),
            receiver-input(m) | m in MESSAGE }
      I = { sender-to-channel(m), channel-to-receiver(m) |
            m in MESSAGE }
  communications
    sender-output(m) | channel-input(m) = sender-to-channel(m)
      for m in MESSAGE
    channel-output(m) | receiver-input(m) = channel-to-receiver(m)
      for m in MESSAGE
  definitions
    System = encaps(H, Sender || Channel || Receiver)
    System' = hide(I, System)
end System
```

In the example we can see that the binding of a parameter is achieved by importing a module *A*, that contains a parameter and binding it to a module *B*. The formal objects from module *A* are bound to the actual objects of module *B* by means of a syntactical construct that resembles the renaming construct. Parameter blocks of an imported module that are not bound, are *inherited* by the importing module. This means that the unbound parameter blocks are added to the set of parameter blocks of the importing module.

Furthermore, this example shows that the definition of a set and the definition of a communication can be parameterized with a variable. Below we give an example of the syntax:

- set definition: $S = \{ a(x) \mid x \text{ in } T \}$
- communication definition: $a(x) \mid b(x) = c(x)$ *for* x *in* T

We should make one final remark about this specification. In the example we bind *message-error*, an element from *MESSAGE*, to the *error* parameter of *Channel*. This means that if the message gets garbled within the channel, *Channel* will use *message-error* to indicate this fact. However, this is not the only situation in which *Receiver* can receive a *message-error*. Because *message-error* is an element of *MESSAGE*, it can also be sent directly by the sender.

1.5 CONCLUSIONS

PSF is a formal specification language suitable for specifying the behaviour of concurrent systems. PSF integrates the process algebra ACP and the algebraic specification language ASF into one language.

PSF specifications consist of two types of modules: data modules, which describe data types, and process modules, which describe process behaviour. Atomic actions and processes can be parameterized with data. The language PSF incorporates several modularization concepts such as: exports, imports and parameters.

Some aspects of concurrent systems cannot be modelled with PSF, such as time dependent behaviour and priorities between actions. However, extensions of the theory ACP contain special operators that deal with real-time [BB91] and priorities [BBK86].†

† Recently, PSF has been extended with *priorities* [Die94].

CHAPTER 2
THE TOOL INTERFACE LANGUAGE

2.1 INTRODUCTION

The Tool Interface Language (TIL) is a low-level language for the specification of concurrent systems, and forms the heart of the PSF Toolkit [Vel93]. A first proposal for TIL was given in [Jac89] and its formal definition can be found in [MV89b] as well as in [Mau91]. In this chapter we will give a description of TIL as it is actually implemented, which is slightly different.

We will explain the language by giving the grammar rules for the different syntactical categories. Furthermore, we will illustrate the use of each syntactical category by means of examples.

2.2 MOTIVATION

TIL was designed as a specification language to be used as an interface between the different tools in the PSF Toolkit. There are several reasons to introduce an intermediate language in the design of a tool set for the analysis of concurrent systems.

An intermediate language clearly supports a layered design in which users can inspect a specification in a high-level language and the tools can operate on a low-level representation. This distinction can be compared to the difference between a conventional programming language directed towards the needs of programmers and a machine-specific assembler language.

Many features of high-level specification languages, such as syntax and modularization techniques, are of no concern to a tool. The effort of writing a complex parser and type checker for the high-level language has only to be taken once. Next, the tools can use a much simpler parser to read the intermediate language. This also indicates that by writing a new front-end, the tool set can easily be adapted to high-level languages with comparable functionality.

To be able to support PSF, TIL must have the same expressive power as the combination of ACP and ASF, from a semantical point of view. Features from PSF that do not add to the expressive power and that are only of importance to humans such as: *modularization*, *overloading*, *renaming* and *parameterization*, are not incorporated in TIL.

To enable the tools to exchange information, and to be able to relate objects in TIL to their original representation in PSF, TIL supports a feature called *free format*. The syntax of the *free format* is not dictated by TIL, but must be agreed on by the tools.

We could have chosen for an approach in which the tools are merely exchanging data structures in their internal (binary) representation. Still we have given preference to an ASCII representation of TIL, because especially during the development phase of the tools, humans frequently have to inspect the generated code. However, because TIL is mainly used by tools, its readability for humans is of secondary importance.

2.3 TIL Syntax Globally

A TIL specification consists of a series of tuples. Such a tuple is a declaration or a definition of one object, for example a *sort*, an *equation*, a *process definition*. The general format of such a tuple is:

key definition free format

The *key* is the identification for an object within the TIL specification. Its general form is: [*X*,*Y*] where *X* is a digit representing the type of the object and *Y* is an integer representing the index in a table of objects of type *X*. Each object is assigned a unique key.

The *definition* field of a tuple defines an object, possibly in terms of other objects. In this field so-called tags, operators and predefined objects can be used. We will explain more about the definition field when we show how the different TIL objects are represented.

The *free format* field is merely a text string. This feature is incorporated to allow for so-called *hooks*. A hook is a piece of information that is used in the process of reproducing a specification in a user-oriented language from a specification in a more computer-oriented formalism. In our case we use the free format field to keep the original names of all objects, but we could as well have registered the line numbers of their appearances in the original specification. Finally, the free format can also be used to exchange information between the different tools. In the current implementation hooks are extensively used to generate legible error messages.

2.4 The Running Example

In this section we will give a PSF version of the specification we will use in the explanation of TIL. The specification deals with a vending machine, selling tea and coffee. This specification is not intended to specify the behaviour of a vending machine in all its detail, but to give an example that contains all relevant aspects of TIL. Therefore, the specification is somewhat unbalanced, meaning that we describe some parts of the behaviour in great detail while we abstract from details in other parts.

2.4.1 Behaviour of the vending machine

The vending machine sells tea for 10 cents and coffee for 25 cents. When a user inserts a coin, the machine activates its appropriate buttons for selecting a drink. If a coin of 25 cents is

inserted both tea or coffee can be ordered. However, if tea is chosen the machine does not give any change. Before any buttons are activated the machine checks whether there are still any cups inside. If there are no more cups it alerts the user and returns the inserted coin.

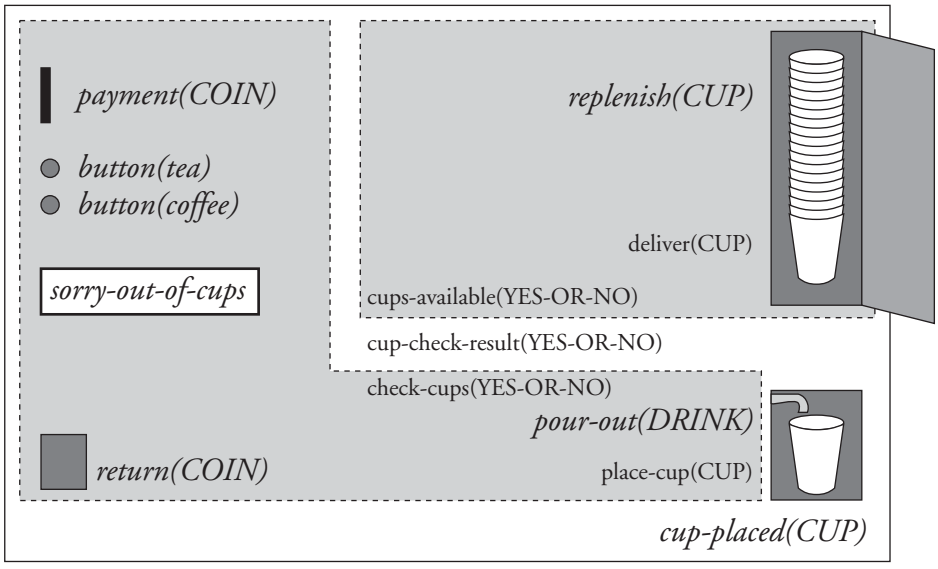


Figure 2.1 Graphical representation of the running example

The vending machine is modelled by two parallel processes, in order to get an example that contains a communication definition. A special process handles the cups inside the machine, while all other behaviour is described by the other process. Figure 2.1 is a suggestive graphical representation of the vending machine. The two different processes are represented by the two large shaded areas. The atomic actions are divided into observable and unobservable actions. The unobservable actions are shown in a smaller font size. Furthermore, the atomic actions that are printed outside the shaded areas, are the result of a communication.

2.4.2 SPECIFICATION OF THE VENDING MACHINE

The specification of the vending machine in PSF is given below.

```
data module VM-Data
begin

  exports
  begin
    sorts
      CUP, STOCK, YES-OR-NO, COIN, DRINK

    functions
      yes :                -> YES-OR-NO
      no  :                -> YES-OR-NO
      5c :                 -> COIN
```

```
      10c  :                -> COIN
      25c  :                -> COIN
      100c :                -> COIN
      coffee :              -> DRINK
      tea    :              -> DRINK
      plastic-cup :         -> CUP
      empty-stock :         -> STOCK
      add : CUP # STOCK     -> STOCK
      cups-in-stock : STOCK -> YES-OR-NO
  end

  variables
    cup : -> CUP
    stock, stock' : -> STOCK

  equations
  [CN] cups-in-stock(stock) = no  when
       stock = empty-stock
  [CY] cups-in-stock(stock) = yes when
       stock = add(cup,stock')

end VM-Data


process module VM-Processes
begin

  imports
    VM-Data

  atoms
    replenish : CUP
    cups-available : YES-OR-NO
    check-cups : YES-OR-NO
    cup-check-result : YES-OR-NO
    deliver : CUP
    place-cup
    cup-placed : CUP
    button : DRINK
    payment : COIN
    return : COIN
    pour-out : DRINK
    sorry-out-of-cups

  processes
    Vending-Machine
    Coin-Slot
    Check-Inventory : COIN
    Get-Order : COIN
    Cup-Supply : STOCK
    Deliver-Cup : STOCK

  sets
    of atoms
      Communication-Atoms =
        { place-cup, deliver(cup) | cup in CUP } +
        { check-cups(yn), cups-available(yn) | yn in YES-OR-NO }
```

```
      Internal-Atoms =
        { cup-check-result(yn) | cup in CUP, yn in YES-OR-NO }
    of COIN
      Valid-Coin =
        { 10c, 25c }

  sets of atoms
    Communication-Atoms =
      { place-cup, deliver(cup) | cup in CUP } +
      { check-cups(yn), cups-available(yn) | yn in YES-OR-NO }
    Internal-Atoms =
      { cup-placed(cup), cup-check-result(yn) |
        cup in CUP, yn in YES-OR-NO }

  communications
    place-cup | deliver(cup) = cup-placed(cup)
      for cup in CUP
    check-cups(yn) | cups-available(yn) = cup-check-result(yn)
      for yn in YES-OR-NO

  variables
    coin : -> COIN
    cup : -> CUP
    stock : -> STOCK

  definitions

    Vending-Machine =
      hide(Internal-Atoms,
        encaps(Communication-Atoms,
          Coin-Slot || Cup-Supply(empty-stock)))
    Coin-Slot =
      sum(coin in Valid-Coin, payment(coin) . Check-Inventory(coin)) .
      Coin-Slot

    Check-Inventory(coin) =
      check-cups(yes) . place-cup . Get-Order(coin) +
      check-cups(no) . sorry-out-of-cups . return(coin)

    Get-Order(coin) =
      [coin = 10c] -> button(tea) . pour-out(tea) +
      [coin = 25c] -> (button(tea) . pour-out(tea) +
                       button(coffee) . pour-out(coffee))

    Cup-Supply(stock) =
      sum(cup in CUP, replenish(cup) . Cup-Supply(add(cup,stock))) +
      cups-available(cups-in-stock(stock)) . Cup-Supply(stock) +
      [cups-in-stock(stock) = yes] -> Deliver-Cup(stock)

    Deliver-Cup(add(cup,stock)) =
      deliver(cup) . Cup-Supply(stock)

end VM-Processes
```

2.5 TIL SYNTAX IN SDF

In this section we will give the grammar of TIL in SDF [HK89] and illustrate the different concepts with examples that are taken from the TIL version of the *Vending Machine* example.

2.5.1 LEXICAL ENTITIES

The following lexical entities are distinguished in TIL:

```
[ \t\n]                                  -> LAYOUT
"--" ~[\n]* "\n"                         -> LAYOUT
"{" ~[}]* "}"                            -> Free-Format
[0-9]+                                   -> Natural
```

All *spaces*, *tabs* and *newlines* are considered layout characters and are discarded in the parsing process. The second entry for *LAYOUT* describes the format of comment in TIL. Comment starts with two hyphens and is closed by a *newline*. We should remark that the current version of the PSF Toolkit does not implement the comment. Free format is defined as a string of zero or more characters, except the '}' character, enclosed by '{' and '}'. Finally, a natural number is a string of one or more digits (0 … 9).

2.5.2 STRUCTURE OF A TIL SPECIFICATION

A TIL specification consists of a series of tuples defining the different objects of a specification. The order of the tuples is irrelevant. The syntax is given by:

```
Entry*                                   -> Specification
Administration-Entry                     -> Entry
Sort-Entry                               -> Entry
Function-Entry                           -> Entry
Atom-Entry                               -> Entry
Process-Entry                            -> Entry
Set-Entry                                -> Entry
Communication-Entry                      -> Entry
Variable-Entry                           -> Entry
Equation-Entry                           -> Entry
Definition-Entry                         -> Entry
```

The syntax shows that a *Specification* consists of zero or more occurrences of *Entry*. The different entries will be explained in the following sections.

2.5.3 ADMINISTRATION

There is one special kind of tuple that consists of a key and a free format field only. These are called administration tuples, and are used to register information that is not specifically related to one item being defined. Its syntax is:

```
Administration-Index Free-Format         -> Administration-Entry
"[0." Natural "]"                        -> Administration-Index
```

The two examples show that the free format is used to register information about the modules that formed the original PSF specification. The convention that is adopted within the PSF Toolkit is, that all information in the free format is preceded by a *tag*, a string of characters between '<' and '>'. The <*m*> tag assigns a number to a module, and a <*t*> tag indicates the original type of the module, *p* for a process module and *d* for a data module.

```
[0.1]   { <m>1 VM-Processes <t>p }
[0.2]   { <m>2 VM-Data <t>d }
```

2.5.4 SORTS

Sorts are also registered using only the key and the free format field. The syntax is:

```
Sort-Index Free-Format                                  -> Sort-Entry
"[1." Natural "]"                                       -> Sort-Index
```

In the example we see that the names of the sorts are registered in the free format using an <*n*> tag. The other tag: <*o*>, indicates the module in which a sort was defined, the module of *origin*. The number refers to the numbers that were assigned to the module names in the *administration tuples*. In this case all sorts were defined in module *2*, *VM-Data*.

```
[1.1]           { <n>CUP <o>2 }
[1.2]           { <n>STOCK <o>2 }
[1.3]           { <n>YES-OR-NO <o>2 }
[1.4]           { <n>COIN <o>2 }
[1.5]           { <n>DRINK <o>2 }
```

2.5.5 FUNCTIONS

We have to make one important remark concerning the syntax of the functions. Currently the PSF Toolkit does not adhere exactly to the official syntax, which is given by

```
Function-Index Natural Sort-Index+ Free-Format          -> Function-Entry
"[2." Natural "]"                                       -> Function-Index
```

The functions are defined by a *Natural* indicating the number of input sorts, followed by one or more instances of *Sort-Index* representing the input sorts and the output sort, in that order.

However, the PSF Compiler still uses an older format that adheres more strongly to ASF, in which it is allowed for a function to have multiple output sorts (*tuples*). This format is given by:

```
Function-Index Natural Sort-Index* Natural Sort-Index+ Free-Format
                                                        -> Function-Entry
```

In this format the first *Natural* indicates the number of input sorts and the second *Natural* the number of output sorts. The tags in the free format have the same meaning as in the previous section.

```
[2.1]   0  1  [1.3]             { <n>yes <o>2 }
[2.2]   0  1  [1.3]             { <n>no <o>2 }
[2.3]   0  1  [1.4]             { <n>5c <o>2 }
[2.4]   0  1  [1.4]             { <n>10c <o>2 }
[2.5]   0  1  [1.4]             { <n>25c <o>2 }
```

```
[2.6]   0  1  [1.4]              { <n>100c <o>2 }
[2.7]   0  1  [1.5]              { <n>coffee <o>2 }
[2.8]   0  1  [1.5]              { <n>tea <o>2 }
[2.9]   0  1  [1.1]              { <n>plastic-cup <o>2 }
[2.10]  0  1  [1.2]              { <n>empty-stock <o>2 }
[2.11]  2  [1.1] [1.2] 1  [1.2]  { <n>add <o>2 }
[2.12]  1  [1.2] 1  [1.3]        { <n>cups-in-stock <o>2 }
```

2.5.6 ATOMIC ACTIONS

The syntax for atomic actions is given by:

```
Atom-Index Natural Sort-Index* Free-Format          -> Atom-Entry
"[3." Natural "]"                                   -> Atom-Index
```

Atomic actions are defined by a *Natural* indicating the number of input sorts, followed by zero or more instances of *Sort-Index* indicating the sort.

```
[3.1]   1  [1.1]        { <n>replenish <o>1 }
[3.2]   1  [1.3]        { <n>cups-available <o>1 }
[3.3]   1  [1.3]        { <n>check-cups <o>1 }
[3.4]   1  [1.3]        { <n>cup-check-result <o>1 }
[3.5]   1  [1.1]        { <n>deliver <o>1 }
[3.6]   0               { <n>place-cup <o>1 }
[3.7]   1  [1.1]        { <n>cup-placed <o>1 }
[3.8]   1  [1.5]        { <n>button <o>1 }
[3.9]   1  [1.4]        { <n>payment <o>1 }
[3.10]  1  [1.4]        { <n>return <o>1 }
[3.11]  1  [1.5]        { <n>pour-out <o>1 }
[3.12]  0               { <n>sorry-out-of-cups <o>1 }
```

2.5.7 PROCESSES

The definition of processes is very similar to the definition of the atomic actions:

```
Process-Index Natural Sort-Index* Free-Format       -> Process-Entry
"[4." Natural "]"                                   -> Process-Index

[4.1]   0               { <n>Vending-Machine <o>1 }
[4.2]   0               { <n>Coin-Slot <o>1 }
[4.3]   1  [1.4]        { <n>Check-Inventory <o>1 }
[4.4]   1  [1.4]        { <n>Get-Order <o>1 }
[4.5]   1  [1.2]        { <n>Cup-Supply <o>1 }
[4.6]   1  [1.2]        { <n>Deliver-Cup <o>1 }
```

2.5.8 SETS

There are several TIL-operators that are used in the definition of sets. The significant part of syntax of the set definition is given by:

```
Set-Index Sort-Index Set-Expr Free-Format           -> Set-Entry
"[5." Natural "]"                                   -> Set-Index
Set-Index                                           -> Set-Expr
"<:," Natural ">(" Enumeration-Item+ ")"            -> Set-Expr
"<+," Natural ">(" Set-Expr+ ")"                    -> Set-Expr
"<.," Natural ">(" Set-Expr+ ")"                    -> Set-Expr
```

```
"<\," Natural ">(" Set-Expr+ ")"                    -> Set-Expr
Term                                                -> Enumeration-Item
Atom-Term                                           -> Enumeration-Item
Atom-Index                                          -> Atom-Term
Atom-Index "(" Term+ ")"                            -> Atom-Term
Variable-Index                                      -> Term
Function-Index                                      -> Term
Function-Index "(" Term+ ")"                        -> Term
"[7." Natural "]"                                   -> Variable-Index
```

The four operators used in set construction are: <:,*n*> (enumeration), <+,*n*> (union), <.,*n*> (intersection) and <\,*n*> (difference). There is no precedence defined on these operators. If brackets are omitted, *left-associativity* is applied.

The *Sort-Index* in the syntax definition of *Set-Entry* indicates the type of the elements of the set. The special *Sort-Index* [*1.0*] is used to indicate that a set is a set of atoms. The example shows the definition of two sets of atoms.

```
[5.1]   [1.0] <+,2>(<:,2>([3.6] [3.5] ([7.6] )) <:,2>([3.3] ([7.1] )[3.2]
        ([7.1])))                          { <n>Communication-Atoms <o>1 }
[5.2]   [1.0] <:,1>([3.4] ([7.1] ))        { <n>Internal-Atoms <o>1 }
[5.3]   [1.4] <:,2>([2.4] [2.5] )          { <n>Valid-Coin <o>1 }
```

2.5.9 COMMUNICATIONS

The communication between atomic actions is described by the *communication function*. The syntax is as follows:

```
Communication-Index Atom-Term Atom-Term Atom-Term Free-Format
                                                    -> Communication-Entry
"[6." Natural "]"                                   -> Communication-Index
```

The communication function is defined by three *Atom-Terms*. The first and second *Atom-Term* indicate the two atomic actions that can communicate with each other. The third *Atom-Term* indicates the atomic action that will be the result of the communication.

```
[6.1]   [3.6]   [3.5] ([7.6] )  [3.7] ([7.6] )          { <o>1 }
[6.2]   [3.3] ([7.1] )  [3.2] ([7.1] )  [3.4] ([7.1] )  { <o>1 }
```

2.5.10 VARIABLES

In TIL there is one kind of variable, as opposed to PSF which employs global variables as well as *placeholders*. Placeholders are used in the definition of sets, communications and the *generalized* versions of the alternative and parallel composition. Both types of variables are registered in the same way in TIL.

```
Variable-Index Variable-Type Free-Format            -> Variable-Entry
"[7." Natural "]"                                   -> Variable-Index
```

The type of the variable being defined is indicated by *Variable-Type*, which can be a reference to either a sort or a set. In the example there is one variable ([*7.5*]) which is typed with a set, the other variables are typed with a sort.

```
[7.1]   [1.3]   { <n>yn <o>1 }
[7.2]   [1.4]   { <n>coin <o>1 }
[7.3]   [1.1]   { <n>cup <o>1 }
[7.4]   [1.2]   { <n>stock <o>1 }
[7.5]   [5.3]   { <n>coin <o>1 }
[7.6]   [1.1]   { <n>cup <o>1 }
[7.7]   [1.1]   { <n>cup <o>2 }
[7.8]   [1.2]   { <n>stock <o>2 }
[7.9]   [1.2]   { <n>stock' <o>2 }
```

2.5.11 EQUATIONS

The syntax for the equational specifications defining the behaviour of the algebraic data types is as follows:

```
Equation-Index Equation Free-Format                    -> Equation-Entry
"[8." Natural "]"                                      -> Equation-Index
Simple-Equation                                        -> Equation
Simple-Equation "<=" Natural {Simple-Equation ","}+    -> Equation
Term "=" Term                                          -> Simple-Equation
```

A non-conditional equation, a *Simple-Equation*, is defined as a pair *Terms* separated by an equal sign ("="). The syntax for conditional equations is given in the second rule for *Equation*. A conditional equation starts with a *Simple-Equation* followed by "<=", a *Natural* indicating the number of conditions and a list of conditions separated by commas. The example shows two definitions of conditional equations.

```
[8.1]   [2.12] ([7.8] ) = [2.2]  <= 1  [7.8]  = [2.10] { <n>[CN] <o>2 }
[8.2]   [2.12] ([7.8] ) = [2.1]  <= 1  [7.8]  = [2.11] ([7.7] [7.9] )
                                                       { <n>[CY] <o>2 }
```

2.5.12 PROCESS DEFINITIONS

The most complex part of a TIL specification is the definition of processes. The relevant part of the syntax is given by:

```
Definition-Index Process-Definition Free-Format     -> Definition-Entry
"[9." Natural "]"                                   -> Definition-Index
Process-Head "=" Process-Expr                       -> Process-Definition
Process-Index                                       -> Process-Head
Process-Index "(" Term+ ")"                         -> Process-Head
Process-Head                                        -> Process-Expr
Atom-Term                                           -> Process-Expr
"<skip>"                                            -> Process-Expr
"<delta>"                                           -> Process-Expr
"<alt," Natural ">(" Process-Expr+ ")"              -> Process-Expr
"<seq," Natural ">(" Process-Expr+ ")"              -> Process-Expr
"<par," Natural ">(" Process-Expr+ ")"              -> Process-Expr
"<encaps>(" Set-Index Process-Expr ")"              -> Process-Expr
"<hide>(" Set-Index Process-Expr ")"                -> Process-Expr
"<sum>(" Variable-Index Process-Expr ")"            -> Process-Expr
"<merge>(" Variable-Index Process-Expr ")"          -> Process-Expr
"<if>" "(" Term Term Process-Expr ")"               -> Process-Expr
```

In TIL the *alternative*, *sequential*, and *parallel* operators (<*alt*,*n*>, <*seq*,*n*> and <*par*,*n*>) can be of any arity, indicated by *Natural*. This is different from PSF where all these

operators are binary. Like the set operators there is no precedence defined, and if brackets are omitted *left-associativity* is applied. The example clearly shows that the TIL syntax is directed towards the interpretation by computers and that some parts of it might be somewhat harder to understand by humans.

```
[9.1] [4.1]  = <hide>([5.2] <encaps>([5.1] <par,2>([4.2] [4.5] ([2.10] ))) )  {
<o>1 }
[9.2] [4.2]  = <seq,2>(<sum>([7.5] <seq,2>([3.9] ([7.5] )[4.3] ([7.5] ))) [4.2]
) { <o>1 }
[9.3] [4.3] ([7.2] ) = <alt,2>(<seq,2>(<seq,2>([3.3] ([2.1] )[3.6] )[4.4]
([7.2] ))<seq,2>(<seq,2>([3.3] ([2.2] )[3.12] )[3.10] ([7.2] ))) { <o>1 }
[9.4] [4.4] ([7.2] ) = <alt,2>(<seq,2>(<if>([7.2] [2.4] [3.8] ([2.8] )) [3.11]
([2.8] ))<if>([7.2] [2.5] <alt,2>(<seq,2>([3.8] ([2.8] )[3.11] ([2.8]
))<seq,2>([3.8] ([2.7] )[3.11] ([2.7] )))) ) { <o>1 }
[9.5] [4.5] ([7.4] ) = <alt,2>(<alt,2>(<sum>([7.6] <seq,2>([3.1] ([7.6] )[4.5]
([2.11] ([7.6] [7.4] )))) <seq,2>([3.2] ([2.12] ([7.4] ))[4.5] ([7.4]
)))<if>([2.12] ([7.4] )[2.1] [4.6] ([7.4] )) ) { <o>1 }
[9.6] [4.6] ([2.11] ([7.3] [7.4] )) = <seq,2>([3.5] ([7.3] )[4.5] ([7.4] )) {
<o>1 }
```

2.6 IMPLEMENTATION

To be able to let the tools access TIL specifications in a uniform way, we have implemented one standard TIL input routine (*parser*). Its definition is given in the file *readtil.c*. This parser reads the input file and produces an internal data structure which is defined in the file *readtil.h*.

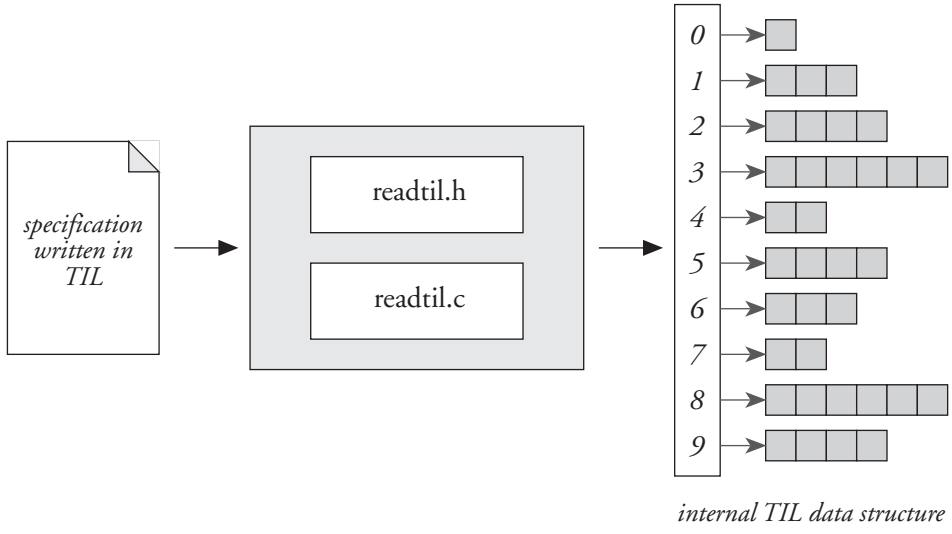


Figure 2.2 The TIL parser

In the design of TIL we have tried to define a language that is easy to parse, such that the parser could be a simple C [KR88] program. This has been done for reasons of efficiency, to save both parsing time and memory space. However, later experiences have shown that

even for such a simple language as TIL, it is beneficial to use a parser generator, like Lex [LS79] and Yacc [Joh79]. The main reasons are that parser generators allow for simpler detection of errors in the input files and easier maintenance.

There are several tools in the PSF Toolkit that generate TIL, most notably the PSF compiler. During the development of these tools we have experienced that it was quite hard at times, to find the errors in the generated TIL code. We think that a parser generator would have given better support than the hand-crafted parser. Problems with the maintenance of the TIL parser showed up when we tried to extend the first version of TIL with the conditional expression (<*if*>). We think that a parser generator also could have facilitated this process.

The way in which TIL objects are defined, by means of *keys* ([*X*.*Y*]) serves one important implementational aspect. It makes it possible to store objects of the same type in an array, and index them by the second entry in the *key*. This provides a direct reference to an object. However, the TIL parser does not know the number of objects of each type in advance, so it guesses a size for the arrays and allocates memory. When there are more objects than there is space in the array, the parser extends the arrays (*realloc*). This approach is appropriate for most specifications we have come across so far. However, in an experiment with a TIL file containing 10,000 process definitions, it showed to be insufficient. There are several solutions to this problem. One possible solution is to make the parsing a two-phase process, where the first phase is only used to count the number of objects of each type. A second solution is to provide information about the number of objects of each type at the top of a TIL file, like the current implementations of M-TIL and I-TIL do.

2.7 CONCLUSIONS

In this chapter we have described TIL, the Tool Interface Language of the PSF Toolkit. An interface language is used to make the toolkit less dependent on the specific high-level language used. This approach guarantees that the toolkit can be made available to other languages based on a combination of algebraic specifications and process algebra, possibly after some changes to TIL. Two other languages, μCRL [GP90] and XP [Vel92], have been interfaced to the PSF Toolkit in this way.

It is possible to extend TIL with new features. The original version did not contain the conditional expression, and it was added without too much effort. We should make a remark here about the extendability. The extensions to TIL must be supported by the different tools in the PSF Toolkit, which might result in complications. We think that extensions such as the conditional expression can be achieved relatively easy. It is still feasible, although somewhat more complicated, to extend TIL and the PSF toolkit with priorities on atomic actions as described in [BBK86] †. However, the introduction of (real-)time aspects [BB91] would probably lead to a redesign of TIL.

† Recently, PSF and the PSF Toolkit have been extended with *priorities* [Die94].

CHAPTER 3
TRANSLATING PSF INTO TIL

3.1 INTRODUCTION

In this chapter we will describe the translation process from PSF to TIL. Although TIL is a clear reflection of the concepts available in PSF, this translation will be performed using two derivatives of TIL called Modular-TIL and Intermediate-TIL. The language M-TIL will be used in the first phase of the translation where all syntactical entities in PSF are translated into a format that resembles TIL but preserves the modular structure present in PSF. In the second phase the main concern is the deletion of the modular structure also called *normalization* or *flattening* and type checking. In this phase I-TIL will be used. Figure 3.1 is a graphical representation of the translation process.

The outline of this chapter is as follows. We start with the description of the translation from PSF into M-TIL. In this section we will illustrate the translation of the syntactical entities using excerpts from PSF specifications followed by their M-TIL equivalents. The next section describes the normalization procedure by means of pseudo-code and introduces I-TIL. The final section describes the translation from I-TIL into TIL. The SDF definitions of the two new languages M-TIL and I-TIL are included as appendices.

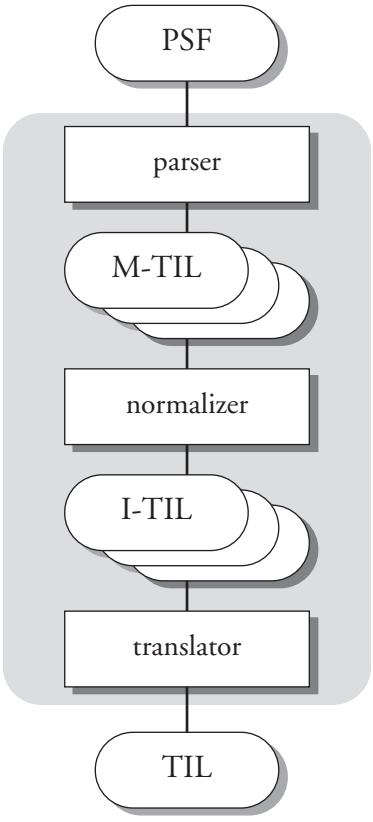


Figure 3.1 The translation from PSF into TIL

3.2 MODULAR TIL

M-TIL serves as an intermediate language between PSF and TIL. Its syntax is similar to TIL, but unlike TIL, M-TIL still contains a modular structure. A first proposal for M-TIL was given in [Jac89].

A specification in M-TIL consists of a series of *modules*. Each module contains a number of *tables* which in turn consist of a series of *tuple* definitions. Each PSF module is translated into exactly one M-TIL module. For all concepts in PSF (*sorts*, *functions*, *atoms*, *processes*, …) there is a corresponding table in M-TIL. The tuples in the M-TIL tables are basically the *declaration* or *definition* of an instance of such a concept. The tuples mentioned here should not be confused with the data structuring mechanism with the same name in ASF.

In the following sections we will treat the translation of data modules, process modules and modularization concepts in that order.

3.2.1 TRANSLATION OF DATA MODULES

In PSF there are two kinds of modules: *data* modules and *process* modules. Data modules give the definition of the data that is used by process definitions in the process modules. We will focus on the translation of data modules first. In all examples in the following section, we will deal with locally defined objects only. The treatment of exported objects is left to the section on modularization.

3.2.1.1 SORTS

There are two kinds of objects in a data module namely *sorts* and *functions*. These objects form the first two tables in an M-TIL specification. We consider the following piece of PSF in which two distinct sorts are introduced:

```
sorts
  BOOLEAN, INTEGER
```

This is translated into:

```
table 1
# 2
[1.1] _ {BOOLEAN}
[1.2] _ {INTEGER}
end
```

From this example we see that the sorts are stored in *table 1*. The *table* keyword indicates the beginning of a table and is followed by a character that tells which concept is defined. On the next line we find a hash (#) followed by a number indicating the number of entries in the table. Then there is one line for each entry. Finally a table is closed by the keyword *end*.

We see that the sort *BOOLEAN* is represented by the *key [1.1]*. The *key* is followed by a *tag* that indicates the visibility of an entity. The tag of the sort BOOLEAN in the example is an underscore (_). The underscore indicates that an object is only visible within the module of its definition. The corresponding tag for an exported item, which is visible outside the module of definition too, is an exclamation mark (!). The free-format, which is enclosed between curly brackets, contains the original names of the sorts.

3.2.1.2 FUNCTIONS

Next we extend the example by introducing some functions that operate on the given sorts.

```
functions
  true : -> BOOLEAN
  false : -> BOOLEAN
  and : BOOLEAN # BOOLEAN -> BOOLEAN
  equal : INTEGER # INTEGER -> BOOLEAN
  …

↓
```

```
table 2
# 9
[2.1] _ 0 1 [1.1]  {true}
[2.2] _ 0 1 [1.1]  {false}
[2.3] _ 2 [1.1] [1.1] 1 [1.1]    {and}
[2.4] _ 2 [1.2] [1.2] 1 [1.1]    {equal)
…
end
```

Functions are defined in *table 2*, so functions have a key starting with *2*. In a function definition the tag is followed by a number indicating the number of input sorts. Next comes a list of references to the input sorts, which can be empty, followed by the number of output sorts and the corresponding list of references.

In PSF it is allowed to use *unary prefix* and *dyadic infix operators* as an alternative for *function* definitions. M-TIL does not support operators, so they are transformed into ordinary prefix functions. The information that these functions were actually operators in the original specification is stored in the free-format by registering the original name of the operator including the underscores indicating the place of the operands. The definition and translation of a prefix operator *not* and an infix operator *and* is given below.

```
functions
  …
  !_ : BOOLEAN -> BOOLEAN
  _&_ : BOOLEAN # BOOLEAN -> BOOLEAN
  …
```

↓

```
table 2
…
[2.5] _ 1 [1.1] 1 [1.1] {!_}
[2.6] _ 2 [1.1] [1.1] 1 [1.1] {_&_}
…
```

3.2.1.3 VARIABLES

Variables in PSF are typed with a sort and are always defined locally. They reside in *table 7* and their translation is straightforward.

```
variables
  x,y : -> BOOLEAN
  …
```

↓

```
table 7
# 3
[7.1] _ [1.1] {x}
[7.2] _ [1.1] {y}
…
end
```

Table 7 is also used to register another kind of variable, the *placeholder*, which we will encounter in the translation of process modules.

3.2.1.4 EQUATIONS

To complete the discussion on the translation of data modules we have to focus on the *equations section*. The equations defining the function *and* are translated as follows:

```
equations
[B1] and(true,x) = x
[B2] and(false,x) = false

↓

table 8
[8.1] [2.3]([2.1] [7.1]) = [7.1] {[B1]}
[8.1] [2.3]([2.2] [7.1]) = [2.2] {[B2]}
```

Equations do not have any tag in their definition field and their translation is accomplished by simply replacing all references to functions and variables by the proper keys and by copying the equation tag from the PSF specification into the free-format. Note that we could have left out the parentheses and equal sign in this section. They are kept, however, to increase readability. Though eventually only tools will use M-TIL, during tool development M-TIL modules will have to be inspected by humans frequently.

PSF also supports conditional equations, which appear in PSF in two alternative forms. In the following example *E* represents an equation.

```
[tag] E1, E2, … , En ==> E
[tag] E when E1, E2, … , En
```

These two forms will have exactly the same representation in M-TIL. Each individual equation that is part of a definition of a conditional equation, is translated as described above and the complete conditional equation is written in M-TIL as follows:

```
[8.X] E <= n (E1 E2 … En)
```

The *n* in the example above indicates the number of equations in the condition.

3.2.2 TRANSLATION OF PROCESS MODULES

Now we will focus on the translation of process modules from PSF into M-TIL.

3.2.2.1 ATOMS AND PROCESSES

The translation of atomic actions and process declarations is similar to the translation of functions. The main difference is that atoms and processes do not have an output type. The example shows that atoms are registered in *table 3* and processes in *table 4*.

```
atoms
  watch-button, serve-coffee, push-button, take-coffee, button-pushed

processes
  Machine, Client, System

↓
```

```
table 3
# 5
[3.1] _ 0       { watch-button }
[3.2] _ 0       { serve-coffee }
[3.3] _ 0       { push-button }
[3.4] _ 0       { take-coffee }
[3.5] _ 0       { button-pushed }
end

table 4
# 3
[4.1] _ 0       { Machine }
[4.2] _ 0       { Client }
[4.3] _ 0       { System }
end
```

3.2.2.2 SETS

A set in PSF is a uniquely named collection of terms of the same sort. There are several ways to construct a set in PSF which are repeated below.

- a sort name represents the set containing all elements of the given sort.
- enumeration of closed terms:
  $\{ t_1, t_2, \ldots, t_n \}$
- enumeration of open terms using placeholders:
  ($\underline{u}$ (= $u_1, u_2, \ldots, u_m$) is a vector of placeholders, D_i is the domain of u_i)
  $\{ t_1(\underline{u}), t_2(\underline{u}), \ldots, t_n(\underline{u}) \mid u_1 \text{ in } D_1, u_2 \text{ in } D_2, \ldots, u_m \text{ in } D_m \}$
- operators on sets: *union, intersection* and *difference*. The set operators in PSF are represented as follows (S_i is a set):
  $S_1 + S_2, \quad S_1 \cdot S_2, \quad S_1 \setminus S_2$

We will continue with an example in which some of the above-mentioned constructions and their translations are given.

```
sets
  of DAY
    Days-of-the-week = DAY
    Weekend = { saturday, sunday }
    Workdays = Days-of-the-week \ Weekend
  of TICKET
    Week-Lunches = { lunch(d) | d in Workdays }
  of atoms
    H = { watch-button , push-button }
    I = { button-pushed }

↓

table 5
# 6
[5.1] _ [1.3] [1.3]               {Days-of-the-week}
[5.2] _ [1.3] <:,2>([2.7] [2.8])  {Weekend}
[5.3] _ [1.3] <\,2>([5.1] [5.2])  {Workdays}
[5.4] _ [1.4] <:,1>([2.9]([7.3])) {Week-Lunches}
[5.5] _ [1.0] <:,2>([3.1] [3.3])  { H }
[5.6] _ [1.0] <:,1>([3.5])        { I }
end
```

The field following the tag field contains the type of the set. That is, the sort to which the elements of the set belong. In PSF it is also possible to declare *sets of atoms*, which are used for constructing *encapsulation* and *abstraction sets*. The special key *[1.0]* which refers to a non-existent sort is reserved to denote a set of atomic actions. Now we will look in turn at each line in the example above.

The first line is the simplest. The set *Days-of-the-week* is represented by the sort *DAY*, which happens to have key *[1.3]*. This key is simply copied and the free-format field is filled with the name of the set.

In the second line we see the enumeration operator: <:,*n*>, where *n* denotes the number of enumerated objects. In this case two constant functions, that represent *saturday* and *sunday* are shown. Note that these two functions are two simple terms. The arguments of an enumeration are in fact translated in the same way as described for terms in the section on equations in data modules.

One of the set operators is shown on the third line. The operator <\,*n*> denotes difference (left-associative), set union is represented by <+,*n*> and set intersection by <.,*n*>. Note that the *n* in these operators denotes the number of arguments. This means that the set operators in M-TIL are not restricted to *binary* operators.

The fourth line shows a simple enumeration (only one argument) in which an open term is used. The variable in this term is called a *placeholder* and will be described in the next section.

3.2.2.3 PLACEHOLDERS

As we saw in the previous section there is a notion of a special type of local variable called: *placeholder*. Apart from the *sets* section, there are other sections in PSF where placeholders can occur, namely the *communication* and *process definition*.

The placeholder is added to the table of the variables but is discriminated from the ordinary variables by giving it a special tag, the '#'-character. Its type can be a sort or a set. Now we extend the previous example by showing the corresponding table of variables.

```
table 7
…
[7.3] # [5.3]  {d}
end
```

3.2.2.4 COMMUNICATION

The communication function that defines which atomic actions can communicate and what the result of this communication will be is given in *table 6* of the M-TIL specification. The definition of a communication always involves three atomic actions. In the representation in M-TIL the first two references denote the two constructing atomic actions and the third reference denotes the result.

Data terms occurring in the communication definition are translated like the terms in the section on equations. Open terms give rise to the use of placeholders. In the communications section the free-format is not used and remains empty.

```
communications
    watch-button | push-button = button-pushed

↓
```

```
table 6
# 1
[6.1] [3.1] [3.3] [3.5]  {}
end
```

3.2.2.5 PROCESS DEFINITIONS

The most complex part of a process module is the section in which the processes are defined by recursive definitions. For the translation of this part we need operators in M-TIL that represent the operators in PSF. These operators have so-called *process expressions* as arguments.

There are two basic process expressions derived from objects we have dealt with before: an atomic action (*[3.x]*) and a process name (*[4.x]*), both possibly parameterized with a data term. Secondly there is one predefined process expression <*i*> (*i* for internal), that represents *skip*, the silent action, from PSF.

Next come the operators for sequential, alternative and parallel composition. In M-TIL we want to be able to take advantage of the associativity of these operators by letting the operators accept a list of process expressions. All operators in M-TIL are written within "<" and ">" symbols, and use one lower case character, which is derived from the first letter of their name, so *s* stands for ***s***equential, *a* for ***a***lternative and *p* for ***p***arallel composition. In the next table we show a PSF expression on the right and its translation on the left. The expression *PE* stands for a process expression.

PE_1 . PE_2 . … . PE_n	→	<s,n>(PE_1 PE_2 … PE_n)
PE_1 + PE_2 + … + PE_n	→	<a,n>(PE_1 PE_2 … PE_n)
PE_1 ‖ PE_2 ‖ … ‖ PE_n	→	<p,n>(PE_1 PE_2 … PE_n)

Besides these operators, PSF supports the generalized alternative and parallel composition, called *sum* and *merge*. These operators use the uppercase counterpart of the characters used in the previous example. Moreover we see the placeholder, used to parameterize the process expression, appear in the translation as the reference *[7.x]*.

```
sum( p in Set_or_Sort, PE(p) )     →    <A>([7.x] PE)
merge( p in Set_or_Sort, PE(p) )   →    <P>([7.x] PE)
```

There are two related operators in PSF that deal with encapsulation and abstraction of atomic actions. These operators take two arguments: a set and a process expression. Their translation is as follows:

```
encaps( Set_of_Atoms, PE )   →    <e>([5.x] PE)
hide( Set_of_Atoms, PE )     →    <h>([5.x] PE)
```

The final process expression is the conditional expression. Its translation is as follows:

[$term_1$ = $term_2$] -> PE	→	<f>([$term_1$ $term_2$ PE)

The translation of a process definition is performed by creating a new entry in the definitions table (*[9.x]*) which is followed by a reference to the process that is being defined (*[4.y]*), an equality sign (=) and the defining process expression. To close this section we will give an example of the translation of some process definitions.

```
definitions
  Machine = watch-button . serve-coffee . Machine
  Client = push-button . take-coffee . Client
  System = hide(I, encaps(H, Machine || Client ))

↓

table 9
# 3
[9.1] [4.1] = <s,2>(<s,2>([3.1] [3.2]) [4.1])   { }
[9.2] [4.2] = <s,2>(<s,2>([3.3] [3.4]) [4.2])   { }
[9.3] [4.3] = <h>([5.6] <e>([5.5] <p,2>([4.1] [4.2])))  { }
end
```

3.2.3 TRANSLATION OF MODULARIZATION CONCEPTS

As mentioned earlier, PSF supports the construction of specifications in a modular fashion. This means that M-TIL must embody concepts that represent *import* and *export* of objects and concepts for defining parameters as well as binding them.

3.2.3.1 EXPORTS

Objects that are to be used in other modules should be made visible to the outside world, that is, they have to be exported. In PSF this is done by introducing a special section in which all exported items are listed. This section is limited to sorts, functions, atoms, processes and sets, so clearly not everything can be exported. In particular it is not possible to export variables.

In M-TIL we follow a different approach. Instead of having a special table for exported items we indicate the visibility of an object by the tag in the definition field. Whenever an item is exported, we indicate this by setting the character in the *tag field* to an exclamation mark (!), as opposed to the underscore (_), which is used to define local or hidden objects.

In the following example we show the translation of a specification with exports:

```
data module Booleans
begin

   exports
     begin
       sorts
         BOOLEAN
       functions
         true : -> BOOLEAN
         false : -> BOOLEAN
         not : BOOLEAN -> BOOLEAN
…

end Booleans

↓
```

```
module Booleans       { d }

table 1
# 1
[1.1] ! { BOOLEAN }
end

table 2
# 5
[2.1] ! 0 1 [1.1]     { true }
[2.2] ! 0 1 [1.1]     { false }
[2.3] ! 1 [1.1] 1 [1.1]     { not }
…
end
```

3.2.3.2 PARAMETERS

In PSF it is possible to define parameterized specifications. The parameters of a module are treated somewhat like the exported objects from the previous section. Parameters in PSF are listed in named *parameter blocks*, and can contain sorts, functions, atoms, processes and sets. Objects from the parameter blocks are registered in the appropriate tables and contain an '@' in their tag field to discriminate them from *hidden* and *exported* items.

To register the parameter block an entry is added to *table P*. The entries in this table consist of a tag, a number indicating the number of objects within the block, the references to all these objects and finally the name of the block which is recorded in the free format.

The example is an excerpt from the specification of a vending machine that is parameterized by a sort *PRODUCT* and a function *price* that gives the price of an item from sort *PRODUCT*.

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

↓

module Universal-Vending-Machine    { p }
…
table P
# 1
[P.1] _ 2 [1.1] [2.1]   { Items-on-Sale }
end

table 1
# 2
[1.1] @  { PRODUCT }
…
```

```
table 2
# 3
[2.1] @ 1 [1.1] 1 [1.2]     { price }
…
end
```

This example shows that the tag field for *Items-on-Sale* is an underscore because the parameter block is defined within this module. The tag field is needed to be able to distinguish a parameter block that is defined in a module from a parameter block that is mentioned in a binding. The latter case will be dealt with in the next section.

3.2.3.3 IMPORTS, RENAMING AND BINDING

The most delicate part in dealing with modular structure in PSF is treated in this section. The three concepts of *import* of modules, *renaming* of objects and *binding* of parameters are intertwined in such a way that they have to be dealt with at once.

Importing a module in PSF can involve renaming and binding. Translation of an import of a module is performed by adding an entry to *table M*, which registers all imported modules, and adding an entry to *table I*, which registers all bindings and renamings involved in an import.

The entry in *table I* consists of a reference into *table M* for the module that is imported, one character from the set {*R*,*B*,*} indicating the order of renamings and bindings and two lists. Both lists start with a number indicating the number of following references. The first list registers the bindings, the second list registers the renamings.

However, in ASF and therefore PSF it is possible to apply a series of renamings to an imported module and perform the binding afterwards or bind the parameters first and then perform the renaming. The two different orders can lead to different results in PSF and so the exact order has to be registered. This is done by the character immediately following the reference into *table M*. When the renaming has to be done first this is indicated by an '*R*', whereas a '*B*' indicates that the binding has to performed first. If there are no renamings or bindings, this is indicated by a '*'. The simplest case of an import is shown below.

```
data module Naturals
begin
  imports Booleans
end Naturals

↓

module Naturals    { d }

table M
# 1
[M.1]   { Booleans }
end

table I
# 1
[I.1] [M.1] * 0 0  { }
…
end
```

A more complex example is the following excerpt from a specification in which a vending machine is instantiated with different kind of drinks.

```
process module VM-Tea-Coffee-Orange
begin

  imports
   Univeral-Vending-Machine {
     Items-on-sale
        bound by [
         PRODUCT -> DRINK
        ] to Drinks
        renamed by [
         get-selection -> watch-button,
         UVM -> VMCTO
        ]
   }
…

end VM-Tea-Coffee-Orange
```

We see that we have to deal with two kinds of renamings in this example. We have the renaming that is part of the binding of *PRODUCT* to *DRINK*, as well as an ordinary renaming.

For all objects that are part of a renaming it is unknown what their type or possible arity is. As a result of this they are all added to *table U*, containing *unknowns*. Furthermore for each renaming there is an entry added to *table R*. Entries in *table R* consist of two references into *table U*, one for the original name and one for the new name.

In this example we see that the binding comes before the renaming so in *table I* we have to put a '*B*' in the appropriate field. We recall that the order of the references to the bindings and renamings in M-TIL is fixed, first bindings then renamings.

To register a parameter binding we have to do several things. First we have to add the parameter block mentioned to *table P*. Because we do not know anything about the block but its name we set its tag field to '?'. Next we add information about the binding in *table B*. Entries in this table contain a reference to a parameter block, followed by a reference to a module, followed by the number of objects that are bound. Because this object to object binding involves a renaming too, we register this by references into *table R*.

```
module VM-Tea-Coffee-Orange      { p }
…

table M
# 3
[M.1]   { Universal-Vending-Machine }
[M.2]   { Drinks }
…

table P
# 1
[P.1] ? 0 { Items-on-sale }
end
```

```
table U
# 10
[U.1]   { PRODUCT }
[U.2]   { DRINK }
[U.3]   { get-selection }
[U.4]   { watch-button }
[U.5]   { UVM }
[U.6]   { VMCTO }
…

table R
# 5
[R.1]   [U.1] [U.2]  { }
[R.2]   [U.3] [U.4]  { }
[R.3]   [U.5] [U.6]  { }
…

table B
# 1
[B.1]   [P.1] [M.2] 1 [R.1]  { }
end

table I
# 1
[I.1] [M.1] B 1 [B.1] 2 [R.2] [R.3]  { }
…

end
```

3.2.3.4 REFERENCES TO INCOMPLETE OBJECTS

As we saw in the previous section, in the example with the parameter block, it is possible in PSF to encounter a reference to an object that is not defined in the module itself. This is, in fact, the case for all objects that are imported. However, it should be possible within M-TIL to refer to such objects in a similar way as we refer to other objects. To meet this problem, the convention is introduced, that whenever an object is encountered that is not defined locally we add it to the appropriate table and label its tag field with an '?'. The example shows a module *B* that uses objects from *Booleans*.

```
data module Values
begin

  imports
    Booleans
  sorts
    VALUE
  functions
    zero : -> VALUE
    one : -> VALUE
    val : BOOLEAN -> VALUE
  equations
  [V1] val(false) = zero
  [V2] val(true) = one

end Values

↓
```

```
module Values      { d }
…
table 1
# 2
[1.1] _  { VALUE }
[1.2] ?  { BOOLEAN }
end

table 2
# 5
[2.1] _ 0 1 [1.1]  { zero }
[2.2] _ 0 1 [1.1]  { one }
[2.3] _ 1 [1.2] 1 [1.1]  { value }
[2.4] ? 0 1 [1.?]  { false }
[2.5] ? 0 1 [1.?]  { true }

table 8
# 2
[8.1] _ [2.3]([2.4]) = [2.1] { [V1] }
[8.2] _ [2.3]([2.5]) = [2.2] { [V2] }

end
```

This example shows that in the definitions of *false* and *true* a special reference *[1.?]* is used. This is a general reference to an *unknown* sort. Because *true* and *false* are not defined locally we are not able to tell their type. We could try to find their type in any of the imported modules, but this would violate the target of *separate compilation*.
There are two quite intricate situations in which it is even impossible to tell to which class an entity belongs. Consider the following examples:

```
process module Strange
begin

  imports
    Mystery-Module
  processes
    X
  definitions
    X = a . B

end Strange
```

Process *X* is defined as the sequential composition of *a* and *B*. But because *a* and *B* are not defined in *Strange* but seem to come from *Mystery-Module* it is not possible to tell whether they are atoms or processes. Both options are possible. To deal with this problem we have chosen to add such objects to *table 4*, the table of processes, and label the tag field with a '$'.

```
module Strange     { p }

table 4
# 3
[4.1] _ 0 { X }
[4.2] $ 0 { a }
[4.3] $ 0 { B }
end
```

```
table 9
# 1
[9.1] [4.1]= <s,2>([4.2] [4.3])  { }
end

end
```

A similar clash can occur between sorts and sets. See the next example:

```
process module Also-Strange
begin
  imports
    Mystery-Module
  sets
    of A-Sort
      A-Set = Set-or-Sort
     of atoms
      H = { r(d) | d in Mystery }
end Also-Strange
```

In this example two sets are defined. The first set which is called *A-Set* is defined to be equal to *Set-or-Sort*. In general *Set-or-Sort* could be either a set or a sort. However in this example we can conclude that it is a set because its type is *A-Sort*. The only sort of type *A-Sort* is *A-Sort* itself, so *Set-or-Sort* must be a set. In the second set that is defined we restrict the argument of atom *r* to *Mystery*. Now we cannot tell whether *Mystery* is a set or a sort. Such objects are added to *table 1*, the table of sorts, with a '$' in the tag field.

```
module Also-Strange   { p }

table 1
# 2
[1.1] ?         { A-Sort }
[1.2] $         { Mystery }
end

table 5
# 3
[5.1] ? [1.0]   { Set-or-Sort }
[5.2] _ [1.1] [5.1]     { A-Set }
[5.3] _ [1.0] <:,1>([3.1]([7.1]))       { H }
end

end
```

3.2.3.5 M-TIL SUMMARY

To conclude this chapter on M-TIL we will give a list of all the possible tables to be found in an M-TIL specification and their ordering within the file.

M modules
P parameter blocks
U undefined items from parameter blocks
R renamings
B bindings

I imports
1 sorts
2 functions
3 atomic actions
4 processes
5 sets
6 communications
7 variables
8 equational specifications
9 process specifications

The tables are strictly separated in two groups. The tables labelled with digits correspond to the same tables in TIL, whereas all tables used to represent the modularization concepts are labelled with characters.

The name of the physical file within the file system is formed by the name of the module and the extension *.mtil*. For example the PSF specification of module *Booleans*, stored in the file *Booleans.psf* is translated into *Booleans.mtil*.

3.3 NORMALIZATION

In the previous section we introduced M-TIL and we showed how a PSF specification can be transformed into a series of M-TIL modules. Because neither of the two semantics on which PSF is based does support any notion of modules, we have to remove all modular structure from the M-TIL specification to be able to attach a semantics to it. The process in which we do so is called *normalization* and the resulting specification is said to be in *normal form*.

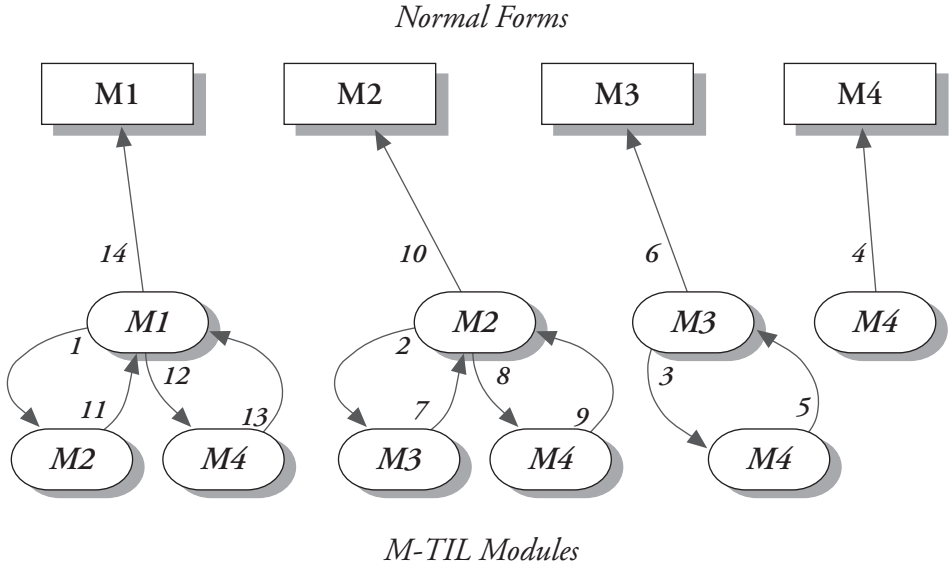


Figure 3.2 Illustration of the normalization procedure

We define the normalization process as follows:

- a module without any imports is in normal form
- to be able to normalize a module *M* with imports, all the imported modules are normalized first. Next, the normal forms of the imported modules and the remainder of *M* are combined producing the normal form of *M*. Possible *renamings* and *parameter bindings* in *M* are encountered during this last step.

Figure 3.2 visualizes this process. The numbers at the arrows indicate the progression of the normalization procedure. At the bottom are the M-TIL modules in the boxes with the rounded corners and at the top the modules in normal form (I-TIL). The two layers of M-TIL modules depict the import relation. For example, *M2* imports both *M3* and *M4*. The target of the normalization illustrated in Figure 3.2 is *M1*. In the following sections we will give the algorithms for normalization.

3.3.1 INTERMEDIATE TIL

The example in Figure 3.2 shows that it is possible to import a module more than once. Module *M1* imports *M2* and *M4*, whereas *M4* is already imported by *M2*. To increase the efficiency of the normalization procedure each module is normalized only once and then stored in an intermediate format. In the example in Figure 3.2 this means that when module *M2* imports *M4*, the normal form of *M4* is already available, as a result of the normalization of *M3*, and does not have to be computed again. In this way, with the aid of an external library manager, which checks whether re-compilation is necessary, the speed of the compilation of a PSF specification can be increased significantly.

Along with the normalization process we have to perform consistency checking. This means that we have to check that all objects that are not defined locally in a module are supplied by one of its imported modules and that their types comply. Moreover, in order to apply the origin rule [BHK89], we must be able to tell the origin (original module) of each object in a specification and check that this origin is unique.

To facilitate the necessary checking we introduce a new format called: I-TIL (Intermediate TIL). A first proposal for I-TIL was given in [Bar90]. I-TIL resembles M-TIL in the way that it is also based on tables containing tuples. I-TIL supports the following tables:

M	modules
P	parameters
1	sorts
2	functions
3	atomic actions
4	processes
5	sets
6	communications
7	variables
8	equational specifications
9	process specifications

The tuples in I-TIL have the same layout as in M-TIL. The only difference is that the key, the pair *[Y.Z]*, is extended with the origin to form *[X.Y.Z]*, where *X* is an index into *table M*

of the I-TIL module at hand. All references to keys in the definition part of a tuple are also extended to include the origin.

It seems that it would have been possible to skip the translation to I-TIL and translate M-TIL directly to TIL. There is however one serious problem in this approach. TIL has been designed to be a low-level language on its own. This means, for example, that because TIL has no support for modularization or parameterization, there is no way to register uninstantiated parameters. Uninstantiated parameters are inherited by an importing module and so they must be represented in the intermediate language. This explains why *table P* is present in I-TIL. Furthermore, we need clear references to the modules that are transformed in the normalization process so that is the reason that *table M* is included in I-TIL.

We could have coded this information about modules and parameters in the free-format fields in TIL, but we think this to be a rather cumbersome solution. Moreover, we want to extend all keys with their origin to simplify the normalization procedure, and so we would get outside the scope of TIL.

3.3.2 THE NORMALIZATION PROCEDURE

In this section we will give the normalization procedure in the form of pseudo-code. This code will be interspersed with ordinary text to clarify the algorithm. We will use the characters *G*, *H* and *J* as variables representing modules.

We think of the normalization procedure as a call to a routine *normalize* that has an M-TIL file (<*name*>*.mtil*) as argument. We start with creating an I-TIL file with the same *base name*, that is, the file name without the extension *.mtil*, as the argument and create the first entry in *table M*. This table will contain one entry for each module that is used in the normalization process. Clearly the current module is at the root of the import tree and so it becomes the first entry. Next we copy all information from the M-TIL file into the I-TIL file but extend all references to keys with their origin, which is entry number *1* in *table M* in this case.

```
normalize(G.mtil)
  // this is comment
  create I-TIL module G.itil
  create a new table M and add first entry with the name G in free-format  //
[1.M.1] { G }
  for every tuple t in tables 1-9 of G.mtil do
    extend the key of t and all references with origin
      and add tuple to G.itil
  od // [X.Y] … [R.S] … -> [1.X.Y] … [1.R.S] …
```

Next we call the normalization routine recursively for every module that is imported. In case of an import with a parameter binding we have to deal with two imported modules. The module to which the parameters are bound is also considered to be an imported module and must be normalized before the parameterized module. In this way, references to parameters in the latter can be resolved with the information already available in the I-TIL module under construction.

```
  for every imported module H in G do
// an actual module in a parameter binding is also considered to be
// imported and must be imported before the module to which it is bound
    if the import contains a binding of the form:
        import H bound to J then
      check consistency of binding list and possible renaming list
      normalize(J.mtil)
      normalize(H.mtil)
     else
      normalize(H.mtil)
    fi
```

To keep track of all the modules that are used in the normalization procedure the list of modules in *table M* of the imported module is merged with *table M* in the module under construction. We have to remember the new index into *table M* for this module to be able to renumber the keys in the tuples of the imported module when they are added to the I-TIL module. This index is used as origin. We also add all parameters, that is, entries in *table P*, to the new I-TIL module. When parameters get bound, these entries will be removed again.

```
    for every tuple t in table M of H.itil do
      if t not in table M of G.itil add t to the end of table M in G.itil
      remember the (old index in H.itil / new index in G.itil) pairs
    od

    for every tuple p in table P of H.itil do
      add p to the end of table P in G.itil
    od
```

All tuples from the imported module will be added to the I-TIL module next. Apart from changing the references to the origins, we have to take care of renamings, possible bindings of parameters, name clashes and so on. The simplest case is when a tuple was local to the imported module. In that case we can simply add it.

```
    for every tuple t in tables 1-9 in H.itil do
      create t' by adjusting the origin in the key of t
        and in all references to keys in the definition part of t
        according to the old/new index pairs

      if t is an object local (tags: _ # $) to module H then
  // comm., equ. and def. are always local
        add t' to the appropriate table
       else   // t is exported from H
```

If an object was not local to the imported module it must have been exported. In the next section of code we deal with the situation in which there was a parameter binding or renaming.

```
        if the import of H contained a parameter binding or
            H is the module that is bound then
          check the order of binding and renaming
          in case there is just a binding do 1:
          in case the order was binding followed by renaming
            do 1:, 2: and then 3:
          in case the order was renaming followed by binding
            do 2: and then 1:
         else
          if the import contained a renaming then
            do 2:
          fi
        fi

1:      if tuple t is a parameter of module H (@ in tag field) then
          if in the binding list the name m of tuple t
              is bound to name n then
            if there is a tuple s with name n and type equal to t
                in the module that is bound then
              remember this t/s pair
             else
              report "parameter cannot be bound"
            fi
           else
            if there is a tuple s with the same name and type as t then
              remember this t/s pair
             else
              remember that t is not bound
            fi
          fi
        fi

2:      if the name in the free format of t is equal to the old name of
          an old name/new name pair from a renaming, change the name in
          the free format of t' into the new name (tables 1-5)
        fi

3:      perform the renaming on the module that is bound
```

Now we have a tuple that has possibly undergone some transformations and we have to check whether it can be added safely to the I-TIL module under construction. Therefore we have to check whether the origin rule is respected. In checking the origin rule we have to match *unknown* tuples with keys containing a '?' (*[1.?]* for an unknown sort) with the defining tuples.

```
        if t' is equal to or can be matched with a tuple already
            existing in G.itil not labelled with a ? or $ tag then
          if origins are equal
            if a match was found change all
                corresponding '?' in the specification
           else
            report "origin rule violation"
          fi
         else
          add t' to the appropriate table
        fi
      fi
    od
```

Now we have merged all tuples from the imported modules into one I-TIL module. The last thing we have to do is to clean up this module. Firstly all parameter blocks that have been bound, and all entries for the formal parameters in these blocks are deleted from the I-TIL module. Moreover it is verified that the bindings were performed properly.

```
  for every parameter-block p to be bound do
// [P.X] ? 0 { name } in G.mtil
    if all elements of p are bound then
      change all references in G.itil to the parameters from p into
        references to the actual objects
      delete all tuples that acted as a parameter
          from p (@ in tag field)
      delete the parameter-block with name p from table P in G.itil
     else
      report "parameter block not fully bound or not bound at all"
    fi
  od
```

Next we have to check whether all objects from *G.mtil* that were supposed to be imported from other modules are really imported. If an object is imported properly, the old entry containing the '?' in the tag field is removed and all references to it are changed. Moreover, we have to resolve the entries in *tables 2* and *4* that were labelled with a '$'. We have to determine the type of an entry (*atom* versus *process* or *set* versus *sort*) and have to remove the old entry.

```
  for every tuple t in table 2/4 do
    if t has $ in its tag field then
      search in tables 1/3 and 2/4 for a tuple s
          with the same name and type
      if exactly one such a tuple s exists then
        change all references to t in all tables of G.itil into
        references to s
       else
        report "unresolved object after evaluating imports"
      fi
    fi
  od

  for every tuple t in tables 1-5 do
    if t is imported (? in tag field) then
      search in the same table for a tuple s
          with the same name and type
      if exactly one such a tuple s exists then
        change all references to t in all tables of G.itil into
        references to s
       else
        report "unresolved object after evaluating imports"
      fi
    fi
  od

od
```

Now we have dealt with all the problems that stemmed from the fact that we have to remove the modular structure from a PSF specification. There are however some other conditions that have to be met and checked:

- *firm handshaking*:
  An atomic action that is the result of a communication cannot occur as a communication partner in another communication.
- *consistency of export*:
  Whenever two atoms that can communicate are exported from a module, the result of the communication must be exported too.
- *consistency of communication*:
  The definition of a communication between two atomic actions must have the same result action in all modules that are combined during the normalization phase.
- well-formedness of terms:
  It must be possible to uniquely type all data terms using *inside-out typing*.
- equation type consistency:
  The left-hand side and right-hand side of an equation must have the same type.
- *scope of variables and placeholders*:
  All variables and placeholders in a process expression must be bound.
  Variables must appear in the parameter list at the right-hand side of the process definition. Placeholders can only be used within the scope of the construction in which they were introduced.

In [MV90,Mau91] these conditions are dealt with in more detail. For a complete example of the normalization procedure we refer to appendix D of [Vel90].

3.4 FROM I-TIL TO TIL

The final step in the translation of PSF specification is the transformation from I-TIL to TIL. This step is relatively simple compared to the normalization procedure.

The main difference between I-TIL and TIL is that in TIL there is no specific order in which the tuples appear, that is, they are no longer grouped in tables. Moreover, the format of the keys in TIL is *[X.Y]* instead of *[X.Y.Z]*. This means that we have to renumber all keys in the final I-TIL specification. This is done by removing the origin from a key and by renumbering the second entry in the new key in increasing order.

The use of the free-format section in TIL is extended. The first use of the free-format is in the administration tuples. These are the tuples in TIL that are marked with a *[0.x]* key and are used to register global information about the specification. The names and types of the modules that were used to normalize a specification are indicated by the <*m*> and <*t*> tags. The <*m*> tag, for *module*, is followed by a number, to be used as an index, and the name of the module. The <*t*> tag, for *type*, is followed by a *d* for a data or a *p* for a process module. This information is necessary for generating proper error messages.

Other tags that appear in the free format are: <*n*> followed by the name of an object, <*o*>, for origin, followed by a number, which is a reference to the index of a module as defined in an <*m*> tag and <*on*> followed by the original name of an object if it has been renamed.

There are two situations in which an I-TIL module cannot be translated into TIL and for which we have to check in the translation. The first is when there are still some unbound parameters left, for TIL does not support parameters. The second is when there are functions that have tuples (multiple sorts) as an output-type. We recall that the *tupled* output type is one of the features of ASF that PSF does not support. We give a description of the final translation in pseudo-code below.

```
if there are still entries in table P then
  report "unbound parameters left"
fi

if there are entries in table 2 containing multiple output sorts then
  report "tuples not supported by TIL"
fi

for every tuple t (= [X.Y.Z]) in table M do
  add tuple [0.Z] to the TIL file, inserting <m>Z in the free format
od

for every tuple t (= [X.Y.Z] …) in tables 1 to 9 do
  calculate Z' the new index for t in the TIL file
  copy t into TIL file as [Y.Z'] …, inserting <o>X in the free format
  remember renumbering [X.Y.Z]/[Y.Z']
od

for every tuple s in the TIL file do
  change all keys of the form [X.Y.Z] into [Y.Z']
      according to the renumbering
od
```

3.5 CONCLUSIONS

In this chapter we have shown the process of translating a specification in PSF into a specification in TIL. In doing so, we have introduced two new languages called M-TIL and I-TIL. We have experienced that working with such ASCII-representable intermediate languages is beneficial in several ways. The language definitions serve as a definition of the interfaces between the tools. Furthermore, examples in the intermediate languages can be generated and modified using conventional text editors.

Especially in constructing the tools for translating PSF into TIL we have come to the conclusion that there are several aspects in PSF that should be changed in future versions of the PSF language.

In this project we have chosen for an approach in which we have tried to realize separate compilation of modules, in an attempt to cut down compilation times for larger specifications. We have found in doing so that the current import/export mechanism is not fit for this approach. In the current setting it is impossible to do any type-checking on a modular basis, and type-checking has to be postponed until the normalization phase. As a result we had to introduce objects labelled with '?' and '$'. This fact not only clutters up the description of the translation in this chapter but surely has its effect on the complexity of the implementation of the parser, type-checker and normalizer.

Although the import/export mechanism has been copied from ASF, the tools for ASF do not encounter the problems described above because they perform no translation on a modular basis but on a complete specification only. To overcome these problems we suggest that future versions of PSF should be equipped with a modularization mechanism that allows for a better definition of the interfaces between modules such that full type information is available on a modular basis, to facilitate type-checking at an earlier stage.

Furthermore, we have experienced that the current parameterization mechanism in PSF is not only at times hard to understand when writing a specification, but also causes severe problems during implementation. In the current version of the normalizer we have solved the known defects from the ASF type-checker written by F. Wiedijk and the ASF compiler written by P. Hendriks [Hen88], but the price to be paid has been a significant increase in the time needed for the construction of this tool. We conclude that future versions of PSF should also be equipped with a different parameterization mechanism, but think that this will need some significant theoretical research into the field of parameterization along the lines of Module Algebra [BHK90] or mechanisms from the domain of object-oriented programming languages [Mey88a].

CHAPTER 4
SIMULATING PROCESS BEHAVIOUR

4.1 INTRODUCTION

Although specifications of small processes can be fully understood by experienced people, it is hard to form a complete mental picture of larger specifications that describe more complex processes. To be able to gain insight in the behaviour of processes, *simulation* of a process has proven to be a useful method.

In this chapter we will describe the functions of a general *simulator* and discuss the implementation of the simulator in the PSF Toolkit. To be able to describe a PSF simulator in general terms we give a description of the process that is responsible for generating possible sequences of atomic actions in a simulator run. These possible actions depend on the process semantics of PSF (ACP). To this end we give a description of the ACP axioms in PSF first. On top of this specification we define the function that transforms a process expression into *head normal form*. Once a process expression is in head normal form, it is easy to separate the possible next actions and rest terms.

After this theoretical part we focus on the actual implementation of the simulator in the PSF Toolkit. We give a description of the user interface and list all possible commands the simulator can handle. Then we give an insight into the implementation of the simulator, we give a functional decomposition of the program and relate the source code files to the different functions. Finally we discuss possible extensions of the simulator.

4.2 GENERAL DESCRIPTION OF A SIMULATOR

The main task of a simulator is to show the possible execution traces of a process. By simulating a process the user gets better insight into its behaviour. Especially deadlocks, livelocks and failing communication schemes are often detected through the use of process

simulation. The reason that specifications contain such errors can often be attributed to the fact that they belong to the more dynamical aspects of modelling (parallel) processes and depend on the interaction between different processes. Careful inspection of process specifications on a local level, within one process or module, will not be able to reveal the problems that can occur at the interface between different processes.

The main problem in determining the behaviour of several interacting processes is the enormous number of possible states of the system one has to consider. A computer tool can assist in doing the bookkeeping and clerical calculations, and leave the interpretation to the user.

A general simulation session starts with identifying the process that has to be simulated. This process determines a certain state S. In the next step the simulator calculates all possible actions, $a_1 \ldots a_n$, that can be executed in S. Then the program displays the actions and lets the user choose one of them. After an action a_i has been chosen, the program determines the resulting new state S' and offers the possible action from S'. This process is repeated until the simulated process has terminated or deadlocked. This is the basic function of a simulator. There are of course many other features that can be added to a simulator to facilitate its use. We will discuss such features in later sections.

From the previous explanation it is obvious that the basic function of a simulator is to calculate the next possible actions that are to be executed from a certain state. This is achieved by transforming each process expression into the so-called *head normal form*. The head normal form of a process is described in more detail in [BW90, p. 31].

4.3 Describing ACP in PSF

To be able to give a formal definition of the procedure that calculates the head normal form of a process expression, we have to define a model of ACP in PSF first. A similar approach has been taken in [Mau89], in order to try to describe ACP in terms of a term rewriting system. There are several differences between that specification and the one we will give in this chapter that we will list below:

- the specification in this chapter is executable as opposed to the one in [Mau89].
- the specification in [Mau89] incorporates data types which have been left out here.
- the specification in [Mau89] defines the ACP axioms directly as equations in the term rewriting system. In our approach we want to *control* the rewriting. This implies that the above-mentioned approach cannot be followed, because the order in which terms are rewritten is implementation-dependent.
  In our approach the evaluation of a process expression explicitly has to be performed by a function that controls the rewrite order. As a result of our approach it is possible to define a recursive (ACP) specification as input term, whereas in the approach in [Mau89] the specification is given directly in terms of the term rewriting system.

A study similar to the topic of this chapter is reported in [Ver92b] where a simulator is described for the language μCRL [GP90]. In [Ver92b] the specification is given in the formalism ASF+SDF [Kli91]. A general theory for simulators is given in [Kor94]. In the following sections we will introduce the data types needed to describe ACP in PSF. In this specification we will use several data types from the PSF standard library such as *Booleans* and *Characters*. The PSF Standard Library is a still evolving set of predefined data types. The first document describing the PSF Standard Library is [Wam93].

4.3.1 CONSTANTS

The first objects we have to define are the constants *deadlock* and *skip*. These constants are of sort *CONSTANT* and are modelled as *delta* and *tau* respectively. In this specification, and subsequent ones, we will define an equality function (*eq*) and a less-than function (*lt*) for the data types we introduce. These functions are necessary to be able to create sets of that data type at a later stage.

```
data module Constants
begin
  exports
  begin
    sorts
      CONSTANT
    functions
      delta : -> CONSTANT
      tau : -> CONSTANT
      eq : CONSTANT # CONSTANT -> BOOLEAN
      lt : CONSTANT # CONSTANT -> BOOLEAN
  end

  imports
    Booleans

  equations
  [EQ0] eq(delta,delta) = true
  [EQ1] eq(delta,tau) = false
  [EQ2] eq(tau,delta) = false
  [EQ3] eq(tau,tau) = true

  [LT0] lt(delta,delta) = false
  [LT1] lt(delta,tau) = true
  [LT2] lt(tau,delta) = false
  [LT3] lt(tau,tau) = false
end Constants
```

4.3.2 IDENTIFIERS

Identifiers are used to attach a name to an atomic action or a process. In this specification we have restricted all identifiers to one-character names. By applying the injection *id*() to a *CHARACTER*, it is turned into an *IDENTIFIER*.

```
data module Identifiers
begin

  exports
  begin
    sorts
      IDENTIFIER
    functions
      id : CHARACTER -> IDENTIFIER
      eq : IDENTIFIER # IDENTIFIER -> BOOLEAN
      lt : IDENTIFIER # IDENTIFIER -> BOOLEAN
  end
```

```
  imports
    Booleans, Characters

  variables
    ch,ch' : -> CHARACTER

  equations
  [EQ0] eq(id(ch),id(ch')) = eq(ch,ch')
  [LT0] lt(id(ch),id(ch')) = lt(ord(ch),ord(ch'))

end Identifiers
```

4.3.3 ATOMS

Atomic actions are modelled by the sort *ATOM*. An atomic action is obtained by applying the injection *atom*() to an identifier.

```
data module Atoms
begin

  exports
  begin
    sorts
      ATOM
    functions
      atom : IDENTIFIER -> ATOM
      eq : ATOM # ATOM -> BOOLEAN
      lt : ATOM # ATOM -> BOOLEAN
  end

  imports
    Booleans, Identifiers

  variables
    id,id' : -> IDENTIFIER

  equations
  [EQ0] eq(atom(id),atom(id')) = eq(id,id')
  [LT0] lt(atom(id),atom(id')) = lt(id,id')

end Atoms
```

4.3.4 PROCESSES

Processes are defined in a way similar to the atomic actions. A process is obtained by applying the injection *proc*() to an identifier.

```
data module Processes
begin

  exports
  begin
    sorts
      PROCESS
    functions
      proc : IDENTIFIER -> PROCESS
      eq : PROCESS # PROCESS -> BOOLEAN
      lt : PROCESS # PROCESS -> BOOLEAN
  end
```

```
  imports
    Booleans, Identifiers

  variables
    id,id' : -> IDENTIFIER

  equations
  [EQ0] eq(proc(id),proc(id')) = eq(id,id')
  [LT0] lt(proc(id),proc(id')) = lt(id,id')

end Processes
```

4.3.5 COMMUNICATION

The fact that two atomic actions are able to communicate is defined in module *Communications*. The function *define* takes as arguments three objects of type *ATOM*, the two partners in communication and the result, and delivers an object of type *COMMUNICATION*. The different atoms can be extracted from such an object by the functions *partner1*, *partner2* and *communication*.

```
data module Communications
begin

  exports
  begin
    sorts
      COMMUNICATION
    functions
      define : ATOM # ATOM # ATOM -> COMMUNICATION
      communication : COMMUNICATION -> ATOM
      partner1 : COMMUNICATION -> ATOM
      partner2 : COMMUNICATION -> ATOM
      eq : COMMUNICATION # COMMUNICATION -> BOOLEAN
      lt : COMMUNICATION # COMMUNICATION -> BOOLEAN
  end

  imports
    Booleans, Atoms

  variables
    atm, atm', atm'' : -> ATOM
    com, com' : -> COMMUNICATION

  equations
  [COM0] communication(define(atm,atm',atm'')) = atm''
  [PT10] partner1(define(atm,atm',atm'')) = atm
  [PT20] partner2(define(atm,atm',atm'')) = atm'

  [EQ0]  eq(com,com') = or (and(eq(partner1(com),partner1(com')),
                                eq(partner2(com),partner2(com'))),
                            and(eq(partner1(com),partner2(com')),
                                eq(partner2(com),partner1(com'))))
  [LT0]  lt(com,com') = true when
           lt(partner1(com),partner1(com')) = true
  [LT1]  lt(com,com') = false when
           lt(partner1(com),partner1(com')) = false,
           eq(partner1(com),partner1(com')) = false
```

```
    [LT2]  lt(com,com') = lt(partner2(com),partner2(com')) when
             eq(partner1(com),partner1(com')) = true

end Communications
```

4.3.6 SETS

A frequently used data type, still missing in the PSF Standard Library, is the *set* data type. In the sequel we will have to specify several types of sets. To reduce specification effort we will give a generic specification of sets in this section. The specification is parameterized by the data type *DATA*.

This specification specifies sets containing unique elements only. Several operations on sets are offered. Sets are created by the function *create*. The function *insert* adds an element to a set and *delete* removes an element. Internally, elements of a set are concatenated by the *&*-operator. The specification includes the usual set operations: *union*, *intersection* and *difference* as well as a function to test whether a set is empty (*empty*) and a function to test whether a set contains a certain element (*element*).

```
data module Sets
begin
  parameters
    Data
    begin
      sorts
        DATA
      functions
        eq : DATA # DATA -> BOOLEAN            -- equal
        lt : DATA # DATA -> BOOLEAN            -- less than
    end Data

  exports
  begin

    sorts
      SET

    functions
      empty-set : -> SET
      _&_ : SET # DATA -> SET
      create : -> SET
      insert : DATA # SET -> SET
      delete : DATA # SET -> SET
      empty : SET -> BOOLEAN
      element : DATA # SET -> BOOLEAN
      union : SET # SET -> SET
      intersection : SET # SET -> SET
      difference : SET # SET -> SET
  end

  imports
    Booleans

  variables
    datum, datum' : -> DATA
    set, set' : -> SET
```

```
    equations

    [CRE0] create = empty-set

    [INS0] insert(datum,empty-set) = empty-set & datum
    [INS1] insert(datum,set & datum') = set & datum  when
             eq(datum,datum') = true
    [INS2] insert(datum,set & datum') = (set & datum') & datum  when
             eq(datum,datum') = false,
             lt(datum,datum') = false
    [INS3] insert(datum,set & datum') = insert(datum,set) & datum'  when
             eq(datum,datum') = false,
             lt(datum,datum') = true

    [DEL0] delete(datum,empty-set) = empty-set
    [DEL1] delete(datum,set & datum') = set  when
             eq(datum,datum') = true
    [DEL2] delete(datum,set & datum') = delete(datum,set) & datum'  when
             eq(datum,datum') = false

    [EMP0] empty(empty-set) = true
    [EMP1] empty(set & datum) = false

    [ELM0] element(datum,empty-set) = false
    [ELM1] element(datum, set & datum') = true  when
             eq(datum,datum') = true
    [ELM2] element(datum, set & datum') = element(datum,set)  when
             eq(datum,datum') = false

    [UNI0] union(empty-set,set) = set
    [UNI1] union(set & datum, set') = union(set, insert(datum, set'))

    [INT0] intersection(empty-set,set) = empty-set
    [INT1] intersection(set & datum, set') =
            intersection(set,delete(datum,set')) & datum  when
             element(datum,set') = true
    [INT2] intersection(set & datum, set') =
            intersection(set,set')  when
             element(datum,set') = false

    [DIF0] difference(set,empty-set) = set
    [DIF1] difference(set, set' & datum') =
            difference(delete(datum',set),set')  when
             element(datum',set) = true
    [DIF2] difference(set, set' & datum') =
            difference(set,set')  when
             element(datum',set) = false

end Sets
```

4.3.6 SETS OF ATOMS

Sets of atomic actions, which are used in the *encapsulation* and *abstraction* operators, are defined by instantiating the generic specification of sets given in the previous section.

```
data module Atom-Sets
begin

  imports
    Sets {
      Data bound by [
        DATA -> ATOM
      ] to Atoms
      renamed by [
        SET -> ATOM-SET,
        empty-set -> empty-atom-set
      ]
    }

end Atom-Sets
```

4.3.7 PROCESS EXPRESSIONS

The process expressions are defined in module *Expressions*. All operators are modelled by a three-character function. Furthermore, there are three different injections *pe*(), which turn atoms, processes and constants into process expressions.

```
data module Expressions
begin

  exports
  begin
    sorts
      EXPRESSION        -- process expression

    functions
      pe : ATOM -> EXPRESSION
      pe : CONSTANT -> EXPRESSION
      pe : PROCESS -> EXPRESSION
      seq : EXPRESSION # EXPRESSION -> EXPRESSION  - sequential comp.
      alt : EXPRESSION # EXPRESSION -> EXPRESSION  - alternative comp.
      par : EXPRESSION # EXPRESSION -> EXPRESSION  - parallel comp.
      lmg : EXPRESSION # EXPRESSION -> EXPRESSION  - left merge
      cmg : EXPRESSION # EXPRESSION -> EXPRESSION  - communication merge
      enc : ATOM-SET # EXPRESSION -> EXPRESSION    - encapsulation
      abs : ATOM-SET # EXPRESSION -> EXPRESSION    - abstraction
  end

  imports
    Atoms, Constants, Processes, Atom-Sets

end Expressions
```

4.3.8 THE COMMUNICATION FUNCTION

The communication function is defined as a set of objects of type *COMMUNICATION*. The specification is obtained by instantiating the generic module *Sets* with the sort *COMMUNICATION*. The function *communication* calculates the result of a communication between two atomic actions. The arguments to *communication* are the communication

function followed by two atomic actions. The result is a process expression. If the communication between the two atomic actions is not defined, the result will be a deadlock (*pe(delta)*).

```
data module Communication-Sets
begin

  exports
  begin
    functions
     communication : COMMUNICATION-SET # ATOM # ATOM -> EXPRESSION
  end

  imports
    Communications, Expressions,
    Sets {
      Data bound by [
        DATA -> COMMUNICATION,
        eq -> eq,
        lt -> lt
      ] to Communications
       renamed by [
        SET -> COMMUNICATION-SET,
        empty-set -> empty-comset
      ]
    }

  variables
    atm,atm' : -> ATOM
    com : -> COMMUNICATION
    comset : -> COMMUNICATION-SET

  equations
  [COM0] communication(empty-comset,atm,atm') = pe(delta)
                                                  -- undefined comm.
  [COM1] communication(comset & com,atm,atm') =
          pe(communication(com)) when
           or (and(eq(partner1(com),atm),eq(partner2(com),atm')),
               and(eq(partner2(com),atm),eq(partner1(com),atm'))) = true
  [COM2] communication(comset & com,atm,atm') =
          communication(comset,atm,atm') when
           or (and(eq(partner1(com),atm),eq(partner2(com),atm')),
               and(eq(partner2(com),atm),eq(partner1(com),atm'))) = false

end Communication-Sets
```

4.3.9 PROCESS DEFINITION

Processes and process expressions are coupled by the function *define* in module *Definitions* to form a process definition. The process or the expression can be extracted from a *DEFINITION* by the functions *process-id* and *process-expression*.

```
data module Definitions
begin

  exports
  begin
    sorts
     DEFINITION
    functions
     define : PROCESS # EXPRESSION -> DEFINITION
     process-expression : DEFINITION -> EXPRESSION
     process-id : DEFINITION -> PROCESS
     eq : DEFINITION # DEFINITION -> BOOLEAN
     lt : DEFINITION # DEFINITION -> BOOLEAN
  end

  imports
   Booleans, Expressions, Processes

  variables
   exp : -> EXPRESSION
   prc : -> PROCESS
   def, def' : -> DEFINITION

  equations
   [PEX0] process-expression(define(prc,exp)) = exp
   [PID0] process-id(define(prc,exp)) = prc

   [EQ0] eq(def,def') = eq(process-id(def),process-id(def'))
   [LT0] lt(def,def') = lt(process-id(def),process-id(def'))

end Definitions
```

4.3.10 SETS OF PROCESS DEFINITIONS

Sets of process definitions are defined by instantiating the generic specification of sets. The function *definition* is used to find the process expression belonging to a process from a set of process definitions. The function *definition* takes as arguments a process and a set of definitions. If a process is not defined in the set, *definition* yields a deadlock (*pe(delta)*).

```
data module Definition-Sets
begin

  exports
  begin
    functions
     definition : PROCESS # DEFINITION-SET -> EXPRESSION
  end

  imports
   Identifiers, Expressions,
   Sets {
     Data bound by [
      DATA -> DEFINITION,
      eq -> eq,
      lt -> lt
     ] to Definitions
```

```
      renamed by [
        SET -> DEFINITION-SET,
        empty-set -> empty-defset
      ]
    }

  variables
    prc : -> PROCESS
    def : -> DEFINITION
    defset : -> DEFINITION-SET

  equations
  [DEF0] definition(prc,empty-defset) = pe(delta)    -- undefined process
  [DEF1] definition(prc,defset & def) = process-expression(def)  when
           eq(prc,process-id(def)) = true
  [DEF2] definition(prc,defset & def) = definition(prc,defset)  when
           eq(prc,process-id(def)) = false

end Definition-Sets
```

4.3.11 SPECIFICATIONS

The final module defines a *SPECIFICATION* as the coupling of a set of process definitions and a communication function by means of the function *specification*.

```
data module Specifications
begin

  exports
  begin
    sorts
      SPECIFICATION
    functions
      specification : DEFINITION-SET # COMMUNICATION-SET -> SPECIFICATION
      process-definition : SPECIFICATION # PROCESS -> EXPRESSION
      communication : SPECIFICATION # ATOM # ATOM -> EXPRESSION
  end

  imports
    Definition-Sets, Communication-Sets

  variables
    defset : -> DEFINITION-SET
    comset : -> COMMUNICATION-SET
    prc : -> PROCESS
    atm,atm' : -> ATOM

  equations
  [DEF0] process-definition(specification(defset,comset),prc) =
           definition(prc,defset)
  [COM0] communication(specification(defset,comset),atm,atm') =
           communication(comset,atm,atm')

end Specifications
```

4.3.12 SUMMARY

Figure 4.1 shows a graphical representation of the relations between the different entities describing the theory for which we will define the algorithm to calculate the head normal form.

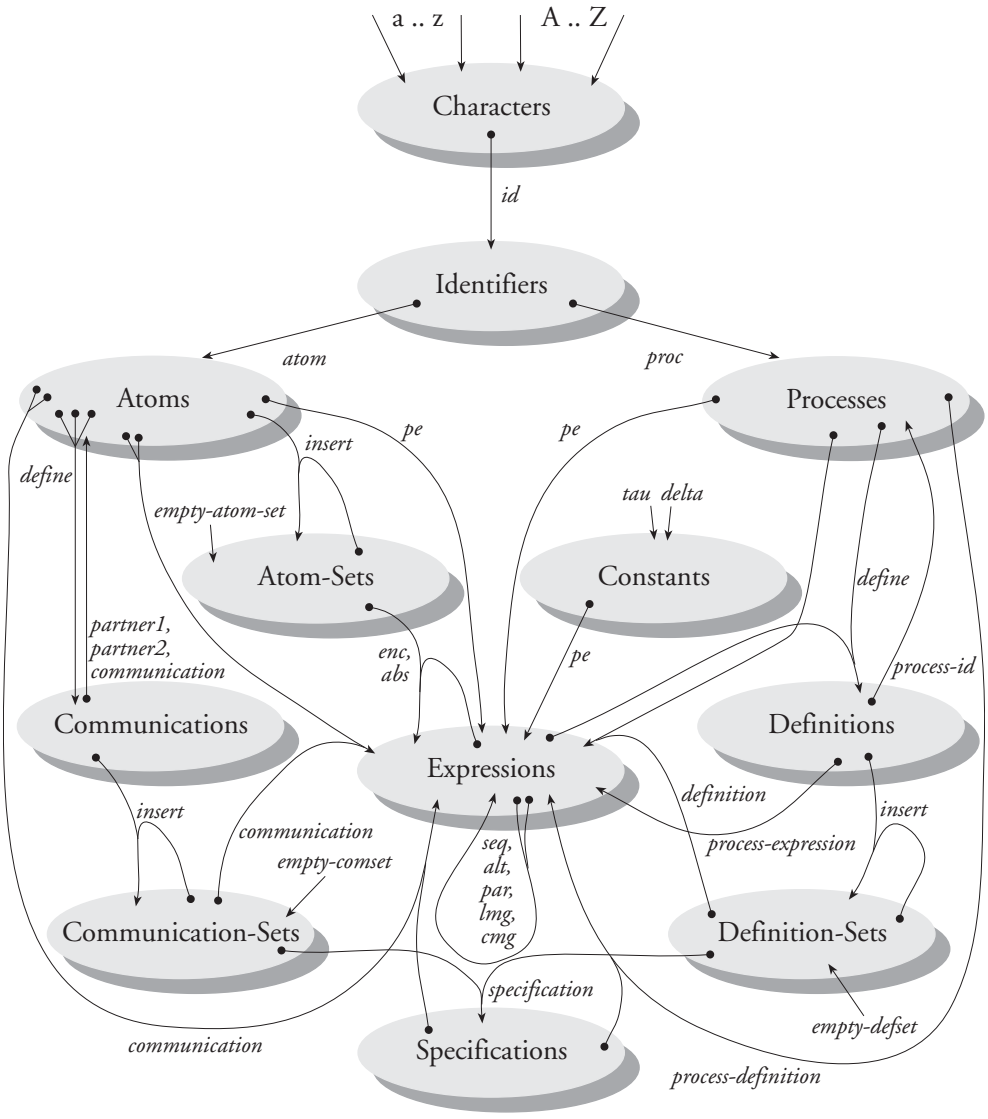


Figure 4.1 Graphical representation of the relations between the different domains

4.4 Calculating the Head Normal Form

In this section we will describe an algorithm that calculates the head normal form of an expression. The algorithm tries to follow an implementation in which the head normal form is constructed with a minimal number of transformations. This implies, for instance, that a process variable is expanded only when it is necessary.

The main function in the following specification is *hnf*. It takes as arguments a specification and a process expression and it yields a process expression in head normal form. There is a special set of *hnf* process operators that are denoted as the normal three-character operators extended with an apostrophe. These operators resemble the normal operators, but they have as argument one or two expressions in head normal form. In the explaining text we will use a more ACP-like notation in which these special *hnf*-operators and process expressions in head normal form will be underlined.

```
data module Normalize
begin

  exports
  begin
    sorts
     HNF-EXPRESSION
    functions
     hnf : SPECIFICATION # EXPRESSION -> HNF-EXPRESSION
     hnf-pe : ATOM -> HNF-EXPRESSION
     hnf-pe : CONSTANT -> HNF-EXPRESSION
     hnf-com : SPECIFICATION # ATOM # ATOM -> HNF-EXPRESSION
     alt' : HNF-EXPRESSION #  HNF-EXPRESSION -> HNF-EXPRESSION
     seq' : HNF-EXPRESSION #  EXPRESSION -> HNF-EXPRESSION
     lmg' : HNF-EXPRESSION #  EXPRESSION -> HNF-EXPRESSION
     cmg' : SPECIFICATION # HNF-EXPRESSION # HNF-EXPRESSION
              -> HNF-EXPRESSION
     enc' : ATOM-SET #  HNF-EXPRESSION -> HNF-EXPRESSION
     abs' : ATOM-SET #  HNF-EXPRESSION -> HNF-EXPRESSION
  end

  imports
    Specifications

  variables
    spec : -> SPECIFICATION
    atm,atm',atm'' : -> ATOM
    as : -> ATOM-SET
    cs : -> COMMUNICATION-SET
    prc : -> PROCESS
    con : -> CONSTANT
    exp,exp' : -> EXPRESSION
    hnf-exp,hnf-exp',hnf-exp'' : -> HNF-EXPRESSION

  equations
```

The following equations hold for the function *hnf*:

hnf(a) = $\underline{a}$
hnf(P) = hnf(def(P))

$hnf(\delta) = \underline{\delta}$
$hnf(\tau) = \underline{\tau}$

To indicate that an atomic action or a constant process expression is in head normal form, the injection *pe*() is replaced by *hnf-pe*(). A process variable is first expanded to its definition and then *hnf* is applied to the result of this expansion.

```
-- atoms, constants and processes
  [HNF00] hnf(spec,pe(atm)) = hnf-pe(atm)
  [HNF01] hnf(spec,pe(con)) = hnf-pe(con)
  [HNF02] hnf(spec,pe(prc)) = hnf(spec,process-definition(spec,prc))
```

The following equations hold for the function *hnf*.

$hnf(x + y) = hnf(x) \underline{+} hnf(y)$
$hnf(x \cdot y) = hnf(x) \underline{\cdot} y$
$hnf(x \between y) = hnf(x \lfloor\!\lfloor y) \underline{+} hnf(y \lfloor\!\lfloor x) \underline{+} hnf(x \mid y)$
$hnf(x \lfloor\!\lfloor y) = hnf(x) \underline{\lfloor\!\lfloor} y$
$hnf(x \mid y) = hnf(x) \underline{\mid} hnf(y)$
$hnf(\partial_H x) = \underline{\partial}_H hnf(x)$
$hnf(\tau_I x) = \underline{\tau}_I hnf(x)$

In PSF this is written as:

```
-- process operators
[HNF03] hnf(spec,alt(exp,exp')) = alt'(hnf(spec,exp),hnf(spec,exp'))
[HNF04] hnf(spec,seq(exp,exp')) = seq'(hnf(spec,exp),exp')
[HNF05] hnf(spec,par(exp,exp')) =
         alt'(alt'(hnf(spec,lmg(exp,exp')),
                   hnf(spec,lmg(exp',exp))),
                   hnf(spec,cmg(exp,exp')))
[HNF06] hnf(spec,lmg(exp,exp')) = lmg'(hnf(spec,exp),exp')
[HNF07] hnf(spec,cmg(exp,exp')) =
         cmg'(spec,hnf(spec,exp),hnf(spec,exp'))
[HNF08] hnf(spec,enc(as,exp)) = enc'(as,hnf(spec,exp))
[HNF09] hnf(spec,abs(as,exp)) = abs'(as,hnf(spec,exp))
```

The function *hnf-com* calculates the result of a communication and delivers a *hnf-expression,* instead of an ordinary expression.

```
-- communication
[HNF10] hnf-com(spec,atm,atm') = hnf-pe(atm'')  when
          communication(spec,atm,atm') = pe(atm'')
[HNF11] hnf-com(spec,atm,atm') = hnf-pe(delta)  when
          communication(spec,atm,atm') = pe(delta)
```

The rewrite rules that implement the standard ACP axioms are only defined on hnf-expressions in order to prevent unnecessary rewriting in sub-terms. As an example we will give the ACP representation for the axioms of the alternative and sequential composition:

$\underline{\delta} \underline{+} \underline{x} = \underline{x}$ $\qquad$ $\underline{\delta} \underline{\cdot} \underline{x} = \underline{\delta}$
$\underline{x} \underline{+} \underline{\delta} = \underline{x}$ $\qquad$ $(\underline{x} \underline{\cdot} y) \underline{\cdot} z = \underline{x} \underline{\cdot} (y \cdot z)$
$\underline{x} \underline{+} (\underline{y} \underline{+} \underline{z}) = (\underline{x} \underline{+} \underline{y}) \underline{+} \underline{z}$ $\qquad$ $(\underline{x} \underline{+} \underline{y}) \underline{\cdot} z = (\underline{x} \underline{\cdot} z) \underline{+} (\underline{y} \underline{\cdot} z)$

```
-- hnf operators
[HNF12] alt'(hnf-pe(delta),hnf-exp) = hnf-exp
[HNF13] alt'(hnf-exp,hnf-pe(delta)) = hnf-exp
[HNF14] alt'(hnf-exp,alt'(hnf-exp',hnf-exp'')) =
         alt'(alt'(hnf-exp,hnf-exp'),hnf-exp'')

[HNF15] seq'(hnf-pe(delta),exp) = hnf-pe(delta)
[HNF16] seq'(seq'(hnf-exp,exp),exp') =
         seq'(hnf-exp,seq(exp,exp'))
[HNF17] seq'(alt'(hnf-exp,hnf-exp'),exp) =
         alt'(seq'(hnf-exp,exp),seq'(hnf-exp',exp))

[HNF18] lmg'(hnf-pe(con),exp) = seq'(hnf-pe(con),exp)
[HNF19] lmg'(hnf-pe(atm),exp) = seq'(hnf-pe(atm),exp)
[HNF20] lmg'(seq'(hnf-pe(con),exp),exp') =
         seq'(hnf-pe(con),par(exp,exp'))
[HNF21] lmg'(seq'(hnf-pe(atm),exp),exp') =
         seq'(hnf-pe(atm),par(exp,exp'))
[HNF22] lmg'(alt'(hnf-exp,hnf-exp'),exp) =
         alt'(lmg'(hnf-exp,exp),lmg'(hnf-exp',exp))

[HNF23] cmg'(spec,hnf-pe(con),hnf-exp) = hnf-pe(delta)
[HNF24] cmg'(spec,seq'(hnf-pe(con),exp),hnf-exp) = hnf-pe(delta)
[HNF25] cmg'(spec,hnf-exp,hnf-pe(con)) = hnf-pe(delta)
[HNF26] cmg'(spec,hnf-exp,seq'(hnf-pe(con),exp)) = hnf-pe(delta)
[HNF27] cmg'(spec,hnf-pe(atm),hnf-pe(atm')) =
         hnf-com(spec,atm,atm')
[HNF28] cmg'(spec,hnf-pe(atm),seq'(hnf-pe(atm'),exp)) =
         seq'(hnf-com(spec,atm,atm'),exp)
[HNF29] cmg'(spec,seq'(hnf-pe(atm),exp),hnf-pe(atm')) =
         seq'(hnf-com(spec,atm,atm'),exp)
[HNF30] cmg'(spec,seq'(hnf-pe(atm),exp),seq'(hnf-pe(atm'),exp')) =
         seq'(hnf-com(spec,atm,atm'),par(exp,exp'))
[HNF31] cmg'(spec,alt'(hnf-exp,hnf-exp'),hnf-exp'') =
         alt'(cmg'(spec,hnf-exp,hnf-exp''),
              cmg'(spec,hnf-exp',hnf-exp''))
[HNF32] cmg'(spec,hnf-exp,alt'(hnf-exp',hnf-exp'')) =
         alt'(cmg'(spec,hnf-exp,hnf-exp'),cmg'(spec,hnf-exp,hnf-exp''))

[HNF33] enc'(as,hnf-pe(con)) = hnf-pe(con)
[HNF34] enc'(as,hnf-pe(atm)) = hnf-pe(delta)  when
          element(atm,as) = true
[HNF35] enc'(as,hnf-pe(atm)) = hnf-pe(atm)  when
          element(atm,as) = false
[HNF36] enc'(as,seq'(hnf-pe(con),exp)) = hnf-pe(con)
[HNF37] enc'(as,seq'(hnf-pe(atm),exp)) = hnf-pe(delta)  when
          element(atm,as) = true
[HNF38] enc'(as,seq'(hnf-pe(atm),exp)) =
         seq'(hnf-pe(atm),enc(as,exp))  when
          element(atm,as) = false
[HNF39] enc'(as,alt'(hnf-exp,hnf-exp')) =
         alt'(enc'(as,hnf-exp),enc'(as,hnf-exp'))

[HNF40] abs'(as,hnf-pe(con)) = hnf-pe(con)
[HNF41] abs'(as,hnf-pe(atm)) = hnf-pe(tau)  when
          element(atm,as) = true
[HNF42] abs'(as,hnf-pe(atm)) = hnf-pe(atm)  when
          element(atm,as) = false
```

```
  [HNF43] abs'(as,seq'(hnf-pe(con),exp)) = hnf-pe(con)
  [HNF44] abs'(as,seq'(hnf-pe(atm),exp)) =
            seq'(hnf-pe(tau),abs(as,exp))  when
             element(atm,as) = true
  [HNF45] abs'(as,seq'(hnf-pe(atm),exp)) =
            seq'(hnf-pe(atm),abs(as,exp))  when
             element(atm,as) = false
  [HNF46] abs'(as,alt'(hnf-exp,hnf-exp')) =
           alt'(abs'(as,hnf-exp),abs'(as,hnf-exp'))
end Normalize
```

4.4.1 AN EXAMPLE

In this section we will give an example of the use of the PSF Toolkit to calculate the head normal form of a simple term. The specification we will use in this example is:

$P = (a \cdot P) \parallel Q$
$Q = b \cdot d$
$R = \partial_{\{a,b\}} P$

The communication function is defined as: $\gamma(a,b) = c$, and we are interested in the head normal form of *R*. This results in the following input term for the term rewriter.

```
hnf(
  specification(
    empty-defset &
    define(proc(id(P)),
      par(seq(pe(atom(id(a))),pe(proc(id(P)))),pe(proc(id(Q))))) &
    define(proc(id(Q)),
      seq(pe(atom(id(b))),pe(atom(id(d))))) &
    define(proc(id(R)),
      enc(empty-atom-set & atom(id(a)) & atom(id(b)), pe(proc(id(P))))),
    empty-comset & define(atom(id(a)),atom(id(b)),atom(id(c)))),
  pe(proc(id(R)))
)
```

The result of rewriting this term is:

```
seq'(
  hnf-pe( atom( id( c))),
  enc((( empty-atom-set & atom( id( a))) & atom( id( b))),
    par( pe( proc( id( P)))), pe( atom( id( d))))))
```

This term translates to: $c \cdot \partial_{\{a,b\}} (P \parallel d)$

4.5 THE USER INTERFACE

In this section we will focus on the user interface of the simulator. The simulator is an interactive program that communicates with the user by way of several windows. The user interface is based on the X Window System. This makes it possible to create a graphical user interface. In this section we will describe the different windows of the simulator and their functionality.

4.5.1 THE PSF WINDOW

The *PSF* window displays a PSF version of the TIL specification that was used as input for the simulator. This translation from TIL to PSF does not yield the original PSF specification because TIL does not support any modular structure. However, PSF demands a division of a specification into data modules and process modules at least. Therefore, the translation consists only of one data module and one process module. All *skip* actions that occur in a specification are extended with a unique number. This is necessary to distinguish between different *skip* actions in the *choose* menu.

This translation is implemented by a program called *til_psf*, which can also be used as a stand-alone program. In the latter situation all names are extended with a number indicating the module of origin, to be able to disambiguate between (local) objects with the same name.

The text in the window may be scrolled with the use of the mouse device, according to the default scrolling possibilities for windows offered by the X Window System. The representation of the complete PSF text resides on a disk file in the file system. This has as effect that scrolling may be slow due to intensive I/O traffic for the disk involved.

4.5.2 THE CHOOSE WINDOW

The *choose* window offers the menu of possible future actions to the user. Initially, after the program has read its input file, the menu displays all processes that are part of the specification. When the user has chosen one of these processes, the program starts with the evaluation of this process. In the next step the menu will consist of all possible initial actions that can be performed by the chosen process. The user can choose one of these actions and the simulator will show the menu with the next possible actions.

This process will continue until there are no more actions to be performed or the user explicitly resets the simulator. In the former case there are two possibilities. The process has either terminated successfully or has reached a deadlock state.

The *choose* window also offers a button (*random once*) which orders the simulator to make a random choice from the menu.

4.5.3 THE FUNCTION WINDOW

The *function* window contains a number of buttons that let the user issue commands to the simulator. There are four categories: *random*, *trace*, *breakpoint* and *control* buttons. We will explain their usage below.

4.5.3.1 RANDOM SIMULATION

We have already explained that the user can choose what the next action will be in any state of the process execution or that the simulator can choose one action itself. The simulator also offers the possibility to let the program choose the next actions continuously. After selecting the *random* button, the simulator does not show the menu any longer but chooses a random execution path until a process terminates or the *random* button is deselected.

The *function* window also contains a button (*random type*) that offers the user a pull-down menu with two different *random modes*: *normal* and *weighted*. The difference between these modes is related to the evaluation of the *sum* operator. We will illustrate this with an example. Consider the following process expression:

sum(x in D, $r(x)$. P) + a

In the *normal* operation mode, the simulator starts with evaluating the *sum* expression and creates an $r(x)$ action in the menu for every element in the (finite) domain of D. Then the single a action is added to the menu and a choice is made. If there are n elements in the domain of D there is a probability of *1* out of *n+1* that the action a is chosen.

In a number of applications this results in an undesired preference for the $r(x)$ actions in the choice of the simulator. The user is mostly interested in either the a action, or any of the $r(x)$ actions. If the random mode is set to *weighted*, the simulator first chooses between the actions a and $r(x)$, and then evaluates a possible *sum*. In this situation the probability to choose the a action has become *0.5*.

4.5.3.2 TRACING

There are two buttons in the *function* window that control the tracing of executed actions. The first button is the *default trace* button. If this (toggle) button is selected, the simulator shows the trace of the process as dictated by its definition. This means that only the visible actions of the process, as governed by the *encapsulation* and *abstraction* operators, are shown.

However, especially during the design phase of a specification, the user can be interested in more detailed information. The *trace* button offers a more flexible way to control the trace information. After selecting this button, the simulator shows a list containing all modules from the original PSF specification, as well as a *quit* button. Selecting the *quit* button ends the *trace* function. However, if one of the modules is chosen, a window is shown containing all items from that module that can be traced. At current these traceable items are atomic actions and processes. The user can select an item by clicking on it with the mouse. Clicking again will deselect an item. The selected items in the list are shown in reverse video.

It is also possible to select a whole group of items. This is achieved by pressing the mouse button on an item and holding it, dragging the mouse to another item, and releasing the mouse button. All items in the rectangle spawn by the press and release point of the mouse are now selected.

4.5.3.3 BREAKPOINTS

There are two buttons in the *function* window that deal with *breakpoints*. There are special situations that can arise during (*random*) simulation, in which the user wants to regain control over the execution. This can be achieved by putting a breakpoint on an atomic action or a process. When the simulator encounters a breakpoint it stops with its execution and shows the menu of possible actions. When the breakpoint mechanism is used for spotting errors in a specification, the simulator can be compared with a source level debugger for a conventional programming language.

Breakpoints are activated by selecting the *breakpoint* button. After this the user can choose items from a list in the same way as described above in the case of selecting trace items.

There is a second button (*breakpoint type*) that shows a pull-down menu when selected. This menu contains three modes for dealing with breakpoints, which depend on the menu of possible actions. Below we will explain these three modes:

- *on execution*
  the simulator halts when it has chosen an item with a breakpoint.
- *stop when one*
  the simulator halts when there is an item with a breakpoint in the menu.
- *stop when all*
  the simulator halts when all items in the menu have a breakpoint.

4.5.3.4 CONTROL BUTTONS

The final three buttons in the *function* window deal with general control of the simulator. The *function* window contains a *reset* button. Selecting this button returns the current simulation to its initial state. The user is offered the menu with processes to select a process to start a new simulation run.

Pressing the *process status* button results in a dump of the internal status of the simulated terms. This information appears in the *message* window. We will explain the format of this dump, in the treatment of the *message* window.

Finally there is a *special* button that offers a pull-down menu on selection. This menu contains three items:

- *save trace*
  save the current contents of the *trace* window to a file. A file name will be asked for.
- *load specification*
  load a new TIL specification. This specification will replace the current one. A file name will be asked for.
- *quit*
  quit the simulator.

4.5.4 THE TRACE WINDOW

All tracing information is written to the *trace* window. The trace is listed as a sequence of actions, one action per line. The *trace* window is a scrollable window so the user can look up and down in this list.

If the action was the result of a communication, this is indicated by the keyword *com.* written before the name of the action, otherwise the keyword is *atom*. If the trace is not due to the default trace setting, the action names are preceded by the keyword: *trace*>.

The information in the *trace* window can be interspersed with messages relating to (un)successful termination of the process or the fact that the simulation has been reset. In the latter case the old information in the *trace* window is retained. The contents of the *trace* window can be saved to a file by the *save trace* menu item offered by the *special* button in the *function* window, as described earlier.

4.5.5 THE MESSAGE WINDOW

This window is used by the simulator to send messages to the user. These messages normally relate to the (un)successful termination of the process or the fact that the simulation has been reset.

The *process status* is also written to the *message* window. The format of this dump is:

```
PID   PPID  STATUS FLAGS HEAD
```

The process id of the (sub)process is listed under *PID*, while *PPID* indicates the process id of the parent process. If the process has children (sub-processes), *STATUS* contains an *S* (for sleeping) followed by the number of children. The field *FLAGS* can contain four different flags:

- T: temporary process
- E: the head atom is encapsulated
- H: the head atom is abstracted
- C: communication with another process is possible

Finally, *HEAD* contains the first atomic action of each sub-process. The following abbreviations are used in this field:

<a> : alternative composition
<s> : sequential composition
<e> : encapsulation
<h> : abstraction

The following example shows a process status dump and its corresponding term represented as a tree. For this example we have used the specification of a simple sender and receiver that are connected by a channel, as given in section 1.4.1.10. The following process status dump reflects the internal structure of the process *System'* in its initial state.

```
PID   PPID  STATUS FLAGS HEAD
  0    ---   S  1        <h>
  1      0   S  2        <e>
  2      1   S  2
  4      2              sender-input
  6      2            E channel-input
  8      1   S  2        <a>
  9      8          T E receiver-error
 10      8          T E receiver-input
```

The corresponding tree representation is given in Figure 4.2.

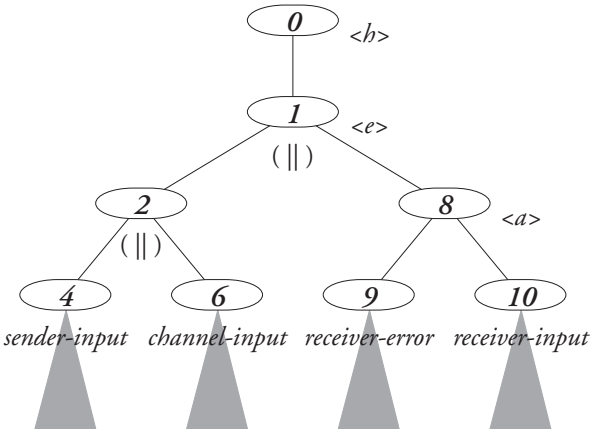


Figure 4.2 Tree representation of process status dump

The current implementation of the simulator does not exactly follow the term rewriting procedure described earlier in an attempt to gain efficiency and reduce memory constraints by trying to avoid the state explosion problem. Instead of rewriting a process expression, a tree of processes is created. All processes receive a process id and can have *children* processes.

The head normal form is constructed dynamically by sending a request downstream the process tree. The processes respond with their initial atoms and send them upstream. During their journey upwards these atoms may come across process operators that determine whether these atoms may travel further upstream. An encapsulation operator may of course block an atom and an abstraction operator can rename it into *skip*.

In this implementation a parallel composition operator is not rewritten into left merges and communication merges but evaluates the streams of atoms coming from its children and determines possible communication actions.

Although this model reduces memory usage by avoiding full expansion and contraction of process terms, it is our experience that it is more difficult to use, from a conceptual point of view, than the pure term rewriting approach. Moreover, changes to the semantics of axioms or additions to the language are rather difficult to implement, because the semantics is hard-coded into the program. In this respect the term rewriting approach leaves more freedom for extensions and changes.

4.6 IMPLEMENTATION

In this section we will briefly sketch the structure of the actual implementation of the simulator. The simulator was implemented in C [KR88] using the X Window System to create a graphical user interface, as was explained before. In Figure 4.3 we present a functional decomposition of the implementation of the simulator.

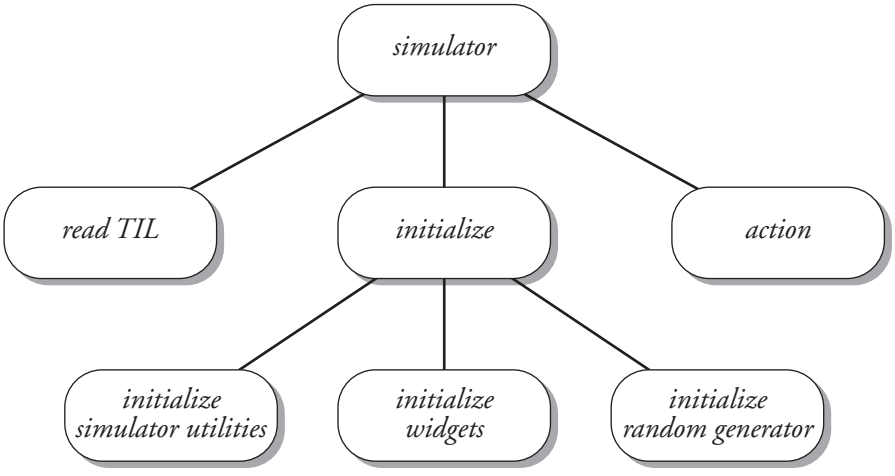


Figure 4.3 Top level functional decomposition

At the top level the simulator is decomposed into three functions. First the TIL input file is read (*readtil.c* | *h*, *readtiltype.c* | *h*, *readexpname.c* | *h*, *fieldex.c* | *h*, *prtilparts.c* | *h*,

tiltype.h) then several initializations are performed and finally the main loop of the simulator is entered in the box labeled with *action*.

The initialization is split up into several domains:

- initialization of utilities that are specific for the simulator *simutil.c | h*
- initialization of X Window entities, widgets and callback routines *widgets.c | h*
- initialization of the random generator *random.c | h*

The functional decomposition of the main loop (*action*) of the simulator is given in Figure 4.4. Module *state control* determines which actions are to be performed (*statecontrol.c | h*). It cooperates closely with the X-event scheduler, which is taken from the X libraries. The X-event scheduler deals with all events that have to do with the user interface such as: registering mouse clicks, resizing and moving windows, moving and changing the cursor.

The second level modules in Figure 4.4 are strongly related to the different windows of the simulator. Module *choose* contains routines to allow (visual) selection of an item from a list (*choosewid.c | h*). These routines are, for example, used in selecting processes or actions.

The random generator, used in making random choices between possible actions, is implemented in *random.c | h*.

The main part of the simulator is implemented in module *simulate* (*simulate.c | h*). The head normal form is computed in this module. Moreover, it contains sub-module *trace* that contains routines that show the actions that are executed (*tracewid.c | h*).

The routines that implement special functions, such as saving traces and loading specifications, are implemented in the *special* module (*specialwid.c | h*).

The final module, *function*, contains routines that implement the buttons (*trace*, *reset*, *process status*, …) from the *function* window (*functionwid.c | h*).

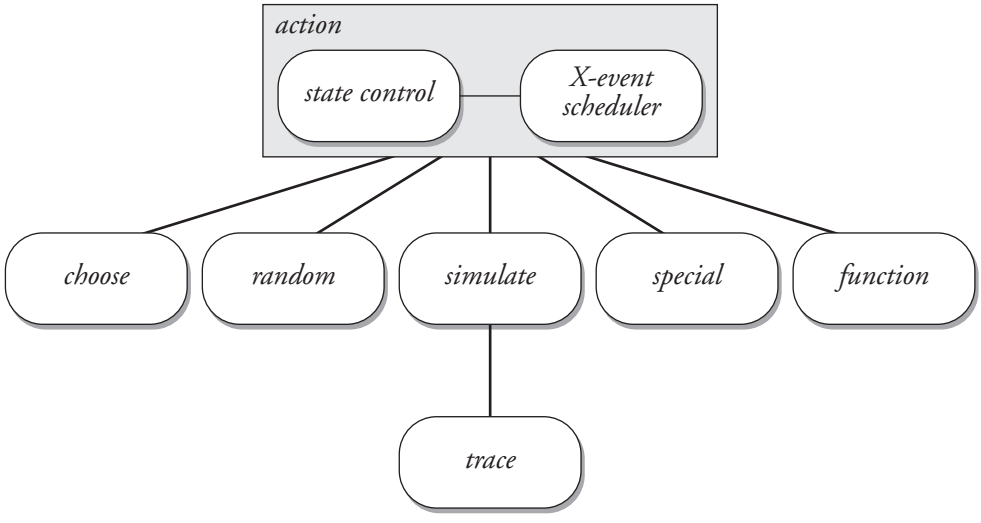


Figure 4.4 Functional decomposition of the main loop of the simulator

4.7 EXTENSIONS

There are several possible extensions of the set of features of a simulator. One of the most wanted features currently not implemented in the simulator is the possibility to *undo* a choice of an atom. With this feature it would be possible to follow the execution of a process for a while and then go back in the execution to a previous state and explore a different path.

To implement this feature there are several approaches feasible. The main point is that the simulator has to remember previous states of the simulation. This can be implemented by maintaining two stacks. The first stack, the *undo* stack, contains all previous states of the simulation run. If the user *undoes* a step the current state is first pushed unto the second stack, the *redo* stack, and the state on top of the *undo* stack becomes the current state. The *redo* operation would do the reverse. First, it pushes the current state on the *undo* stack and then it pops the current state from the *redo* stack.

Because the number of states that are visited during a simulation run can grow very large very quickly, an implementation might choose to retain a limited number of previous states on its *undo* stack. The size of the *redo* stack is implicitly bound by the size of the *undo* stack.

A different direction of extensions is related to the visual representation of the simulation process. User interfaces of computer programs are becoming more and more graphically oriented. Although this is a mere issue of representation of output and not an extension of the functionality of a simulator, we think that future generations of such programs will (have to) pay more attention to this aspect.

Especially the modelling of parallel processes can be represented better by a set of graphical objects, connected to each other by communication channels, than by large and complex expressions in a language such as PSF. The interaction between different processes can also be represented more clearly in a graphical environment, because a communication can be shown as an event appearing on a communication channel between processes. This identifies the process interaction in a more intuitive way than by just listing the name of a communication action.

4.8 CONCLUSIONS

In this chapter we have discussed the concepts that support the simulation of process behaviour. We have shown how the construction of the head normal form of a process expression is the basis for any simulator tool.

To give a formal specification of the process of constructing a head normal form of an expression we have first given a definition of ACP in PSF. On the basis of this ACP specification we have defined the function *hnf()* that turns an ACP expression into an ACP expression in head normal form.

After this more theoretical discussion we have described the simulator as it is implemented in the PSF Toolkit. First we have given a description of the graphical user interface and then sketched the implementation by giving the functional decomposition of the simulator and the related C source files.

The most important conclusion concerning the current simulator is that the implementation of a process simulation tool would benefit from a systematic term rewriting approach. Although the performance might be somewhat worse, the conceptual framework is clearer and the maintainability and extendability of the software is significantly increased. A first start with a re-implementation of the simulator in this direction has been made with the coding of the transition system generator, which will be discussed in Chapter 6.

Finally we have given a number of possible extensions to process simulating tools. These extensions fall largely into two categories; better navigation means to allow for more flexible analysis of the (complete) state space and the representation of output of a simulator in a more graphical-oriented fashion.

CHAPTER 5
A PROOF ASSISTANT FOR PSF

WITH S. MAUW

5.1 INTRODUCTION

One of the advantages of the use of formal techniques for the specification of parallel systems is that it enables formal verification of the correctness of such a specification. There are several approaches towards verification. One can verify certain properties of a specification, such as deadlock-freedom, fairness or starvation-freedom. A more general approach is to verify the truth of logical propositions about the execution traces of a specified system, see for example [HM85,TT94]. We will focus on a third approach, namely verifications of equality of two specifications, as developed in [Bae90]. Equality in this context can be interpreted in many ways, depending on the desired semantics.

A common way of proving that two processes are equal is by interpreting (or defining) the processes in some model, typically a graph model, followed by testing whether the interpretations are equal with respect to some congruence relation, such as observational equivalence or (rooted) weak bisimulation. For finite process graphs several more or less efficient algorithms have been developed for determining these congruences [FM91,GV90]. All of these algorithms suffer from the so-called *state explosion* problem. This problem comes from the fact that the number of states in a complex system is proportional to the product of the number of states of its parallel components.

An alternative is the algebraic or axiomatic approach, where a process expression is manipulated and proven equal to another process expression at a syntactic level, using an effectively given set of axioms. The advantage of this method over exploring the state space is that one can reason about the components or subsystems at a higher level of abstraction. Subsystems can be replaced by simpler ones and this way of pruning in the state

graph results in simpler proofs. Another advantage is that an algebraic approach gives more insight into the reasons why a proof works or fails. This might give clues as to how faulty specifications can be repaired, and how correct specifications can be optimized. The axiomatic approach is also used in the PAM project [Lin92].

The restriction to finite state machines, or the class of regular processes, that is implied by the state exploration methods does not apply to the axiomatic approach. This way more complex processes, such as an unbounded queue, can be considered. The main drawback of an axiomatic approach is that an equality guaranteed by some state space exploration algorithm need not be effectively constructable in the axiomatic system.

Computer tools supporting the axiomatic approach can be divided into two classes: theorem provers and proof assistants. The distinction is based on the level of mechanization of the process of proving. A simple proof assistant will have the form of an "electronic notebook" with accompanying software, which helps in rewriting the formulas that constitute the proof and will depend heavily on the interaction between people and machines. A more sophisticated theorem prover would make use of a number of heuristics to decide automatically what axioms to apply in what order.

In this chapter we will describe the proof assistant for PSF. This tool will be an aid in editing process expressions, selecting axioms that are applicable and applying these axioms. Preferably, sequences of applications of axioms that are commonly used in proofs must be offered using some shorthand. We will call these sequences: *tactics*.

This chapter is organised in the following way. We start with a description of what we consider a proof within the proof assistant, give the axioms that are used to construct proofs and discuss the tactics that have been implemented. Then the user interface of the proof assistant and its implementation are described. We conclude with an example to demonstrate the usage of the proof assistant.

5.2 PROOFS

In this section we will shortly discuss what we consider a proof in the setting of the proof assistant. A verification of a process expression P consists of a stepwise transformation of P into another process expression P'. This transformation can be seen as a proof that the expressions P and P' are equal in a certain process semantics. The semantics depends on the axioms used within the proof assistant. In the current implementation we use the observation equivalence (τ-bisimulation) semantics [BW90].

In the next section we will give the axioms used by the proof assistant. If every step in a proof can be motivated by one of the given axioms, it can be considered correct. Using this technique, correctness of a communication protocol for example, is demonstrated by constructing a proof that the protocol specification and the service specification denote the same process.

Possible computer support in the construction of proofs can be expected in the following areas:

- the editing of an expression
- the suggestion and selection of suitable axioms
- the application of trivial transformation sequences
- the report generation of a proof
- the process of checking manually constructed proofs

5.3 AXIOMS

In this section we will define the axioms that are implemented in the proof assistant. Because these axioms are based on the axiomatization of ACP, we have to explain some ACP notions that are not incorporated in PSF.
The communication function in ACP is defined on the set of atomic actions A as a fixed partial function $\gamma : A \times A \to A$.
The parallel composition in ACP is defined in terms of two additional operators:

- $\between$, left merge.
  $x \between y$ is the process that represents the simultaneous execution of x and y in which the first action to be performed must come from x.
- $\mid$, communication merge.
  $x \mid y$ is the process that represents the simultaneous execution of x and y in which the first action to be performed must be a communication between an action from x and an action from y.

In [AB91] it is shown that the axioms for ACP can be transformed into a complete term rewriting system modulo commutativity and associativity. The axiom system we will present in this section follows the approach of [AB91], but is not a complete term rewriting system because commutativity and associativity are included.

The basic interpretation of an axiom in the proof assistant is a rewrite rule in which the left side is rewritten into the right side. However, during construction of a proof, it is sometimes necessary to interpret an axiom as a rewrite rule from right to left, to allow for full power of expression. This fact has of course as result that an evaluation can result in an infinite rewriting of terms. It is the responsibility of a user of the system to avoid cycles in a calculation.

In the axioms we have given preference to the left-associative form of an expression, see for example axiom ALT_ASSOC. An exception to this rule is the representation of the sequential composition. Here the right-associative form is preferred, see for example SEQ_ASSOC, because one is mostly interested in the first atom (head) of a sequential composition.

Some rules are added that are not present in the standard ACP axiomatization. They merely serve to optimize the rewriting within the proof assistant. See DLK_MRG for an example of an equality that, although it can be derived from the basic axioms, is a useful property to shorten proofs.

$x + y = y + x$	ALT_COMM
$x + (y + z) = (x + y) + z$	ALT_ASSOC
$x + x = x$	ALT_IDENT
$x + \delta = x$	DLK_ALT
$(x + y) \cdot z = x \cdot z + y \cdot z$	SEQ_ALT
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	SEQ_ASSOC
$\delta \cdot x = \delta$	DLK_SEQ

$x \between y = y \between x$		MRG_COMM
$x \between (y \between z) = (x \between y) \between z$		MRG_ASSOC
$x \between y = (x \lfloor\!\lfloor y + y \lfloor\!\lfloor x) + x \mid y$		MRG_LMRG
$x \between \delta = x \cdot \delta$		DLK_MRG
$a \lfloor\!\lfloor x = a \cdot x$		LMRG_SEQ
$a \cdot x \lfloor\!\lfloor y = a \cdot (x \between y)$		LMRG_SEQMRG
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$		LMRG_ALT
$x \lfloor\!\lfloor \delta = x \,.\, \delta$		LMRG_DLKSEQ
$\delta \lfloor\!\lfloor x = \delta$		DLK_LMRG
$a \mid b = \gamma(a,b)$	if γ is defined	CMM_DEF
$a \mid b = \delta$	if γ is undefined	CMM_UNDEF
$x \mid \delta = \delta$		DLK_CMM
$x \mid y = y \mid x$		CMM_COMM
$(a \cdot x) \mid b = (a \mid b) \cdot x$		CMM_SEQ
$(a \cdot x) \mid (b \cdot y) = (a \mid b) \cdot (x \between y)$		CMM_SEQMRG
$(x + y) \mid z = x \mid z + y \mid z$		CMM_ALT
$\partial_H(a) = a$	if $a \notin H$	ENC_ATM
$\partial_H(a) = \delta$	if $a \in H$	ENC_DLKATM
$\partial_H(\mathit{skip}) = \mathit{skip}$		ENC_SKP
$\partial_H(\delta) = \delta$		ENC_DLK
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$		ENC_ALT
$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$		ENC_SEQ
$\tau_I(a) = a$	if $a \notin I$	HID_ATM
$\tau_I(a) = \mathit{skip}$	if $a \in I$	HID_DLKATM
$\tau_I(\mathit{skip}) = \mathit{skip}$		HID_SKP
$\tau_I(\delta) = \delta$		HID_DLK
$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$		HID_ALT
$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$		HID_SEQ
$x \cdot \mathit{skip} = x$		SKPACP_T1
$\mathit{skip} \cdot x + x = \mathit{skip} \cdot x$		SKPACP_T2
$a \cdot (\mathit{skip} \cdot x + y) = a \cdot (\mathit{skip} \cdot x + y) + a \cdot x$		SKPACP_T3
$x \cdot (\mathit{skip} \cdot y) = x \cdot y$		SKPACP_T1B

At the end of the table we have added (one of the possible sets of) laws for *skip*, known as Milner's τ-laws [Mil80]. These axioms define the bisimulation semantics as *observational congruence*. Technically speaking the last axiom is not necessary, because it is a consequence of the first axiom. However, we have added it because of the right-associative form used for the sequential composition.

5.4 TACTICS

Trying to prove facts by using only the axioms provided by the proof assistant can be a tiresome job and therefore error-prone. As a typical example we found in one of our first experiments with the initial implementation of the proof assistant, that a simple proof that takes seven steps when done with pencil and paper, takes more than sixty steps when applying only one axiom at a time. It goes without saying that a successful proof assistant should provide means to shorten such proofs.

We have tried to cope with this problem by trying to mimic the reasoning used by human provers. In doing this, however, we remain exact all the time, that is, we do not want to rely on *heuristics* of any kind. In this section we will discuss some of the tactics that we have found and that are implemented in the proof assistant. These tactics were developed by analyzing a number of manual ACP verifications from [Bae90].

5.4.1 REMOVING DEADLOCKS

To keep up with the state explosion problem it is crucial to be able to prune the state space as soon as possible. One of the strategies to follow is to try to produce and remove deadlocks as early as possible. By trying to apply the axiom $delta \cdot x = delta$ as soon as possible in the rewriting process, we can prevent unnecessary rewriting within x.

Deadlocks are created, for example, within encapsulation expressions when atomic actions are prohibited to occur because they are blocked. A blocked atomic action could escape the blocking when it can engage in a communication and get renamed. However, atomic actions that are blocked by an encapsulation operator and are not able to communicate can be renamed into deadlocks safely. The strategy that creates such deadlocks is called *find deadlocks* in the proof assistant.

A related strategy is *eliminate deadlocks*. This strategy removes, possibly multiple, deadlocks in one step by applying the axiom: $delta + x = x$, to the complete process expression.

5.4.2 HEAD-TAIL NORMALIZATION

The next strategy is one of the main strategies humans apply in constructing proofs in an ACP setting. It is related to computing the *head normal form* of an expression. In this strategy we try to separate a term X into a *head*, the first atom that is possible to occur, and the rest of the term, its *tail*. In analogy this strategy is called *head-tail*. In general the resulting term is not simply a head followed by a tail but it is of the form: $h_1 \cdot t_1 + h_2 \cdot t_2 + \ldots + h_n \cdot t_n$. In trying to create the head-tail expression each process variable that is encountered, is expanded. This means that the lefthand-side of a process definition is replaced by the appropriate righthand-side.

In fact this head-tail strategy is a combination of a number of axioms that relate closely to the expansion theorem and head normal form from [BW90]. For brevity we state this axiom in the case we have a merge with two components X and Y, defined by

$X = a_1 \cdot X_1 + \ldots + a_n \cdot X_n$
$Y = b_1 \cdot Y_1 + \ldots + b_m \cdot Y_m$, *then*

$$X \between Y = a_1 \cdot (X_1 \between Y) + ... + a_n \cdot (X_n \between Y) + b_1 \cdot (X \between Y_1) + ... + b_m \cdot (X \between Y_m) + (a_1 \mid b_1) \cdot (X_1 \between Y_1) + (a_1 \mid b_2) \cdot (X_1 \between Y_2) + ... + (a_n \mid b_m) \cdot (X_n \between Y_m)$$

In case the merge is surrounded by an encapsulation operator, we have the following:

$$\text{encaps}(H, X \between Y) = \sum_{\{i \mid a_i \notin H\}} a_i \cdot (X_i \between Y) + \sum_{\{j \mid b_j \notin H\}} b_j \cdot (X \between Y_j) + \sum_{\{i,j \mid a_i \mid b_j \notin H\}} (a_i \mid b_j) \cdot (X_i \between Y_j)$$

As a corollary to the head-tail strategy the proof assistant implements the *recursive head-tail* strategy. This strategy should be used only on expressions with finite evaluations, otherwise the tool will wind up in an endless recursion.

5.4.3 CONDITIONAL AXIOMS

Finally there are three strategies implemented that relate to the so-called conditional axioms [BBK87]. These axioms are very useful in breaking a complex specification down into subsystems. They support a modular approach towards verification.

Before giving the axioms we have to introduce the notion of the alphabet $\alpha(x)$ of a process x. This is the collection of all atomic actions that process x can perform (see [BBK87] for a definition).

Since this notion is not decidable (see [BBK87]), the alphabet of process x will be approximated by the collection of all actions used in the specification of x or one of its sub-processes as well as all the results of all possible communications between these actions. This inexactness does not influence the validity of the axioms.

Another notation which will be used is the communication set $S \mid T$ of two sets of atoms, S and T. This is defined by

$$S \mid T = \{ a \mid b \mid a \in S, b \in T\}.$$

The first conditional axiom implemented in the proof assistant deals with pushing encapsulations through a merge.

$$\text{encaps}(H, X \between Y) = \text{encaps}(H, X \between \text{encaps}(H', Y)),$$
$$\text{where } H' = (H - \{ a \in \alpha(Y) \mid (\{a\} \mid \alpha(X)) \cap H^C) \neq \emptyset\}) - \{a \mid a \notin \alpha(Y)\}.$$

The set H' is derived from H by first deleting all elements that can take part in a communication of which the resulting action is not encapsulated. Secondly we delete the actions from H that are superfluous because they do not occur in Y.

The second conditional axiom deals with pushing a hide through a merge.

$$\text{hide}(I, X \between Y) = \text{hide}(I, X \between \text{hide}(I', Y)),$$
$$\text{where} \quad I' = (I - \{ a \in \alpha(Y) \mid (\{a\} \mid \alpha(X)) \neq \emptyset\}) - \{a \mid a \notin \alpha(Y)\}.$$

The third axiom is a combination of the first two.

$$\text{hide}(I, \text{encaps}(H, X \between Y)) = \text{hide}(I, \text{encaps}(H, X \between \text{hide}(I', Y))),$$
$$\text{where} \quad I' = ((I - H) - \{ a \in \alpha(Y) \mid (\{a\} \mid \alpha(X)) \neq \emptyset \}) - \{a \mid a \notin \alpha(Y)\}.$$

These three axioms are easily proved correct for closed process expressions, using the conditional axioms from [BBK87].

5.4.3.1 AN EXAMPLE

The following example will clarify the use of these axioms. We consider an array of n components C_i, serially connected to each other. The components can communicate with their neighbours. This system is modelled by a parallel composition of all components:

hide(I, encaps(H, C1 || C2 || … || Cn))

Without giving a description of the behaviour of the components, we assume that each component has k states. Thus the parallel composition (before encapsulation and abstraction) has k^n states. However, the number of states in the system is reduced when encapsulation and abstraction are applied. This is due to the fact that a large number of states become *unreachable*, for example because a communication leading to such states is renamed into deadlock by the encapsulation. So if we would evaluate the system of components in a naive way, starting with the individual C_i components, followed by the elaboration of the parallel composition, we would be doing too much work by taking into account states that later turn out to be unreachable.

By applying the conditional axioms described above, we can focus on the subsystem consisting of the first two components, which has k^2 states, and reduce it to a smaller system. The following step would be to focus on the subsystem obtained by combining this newly derived system and the third component. This process will continue until all components have been merged. The result of this operation is that by restricting oneself to sub-systems only, the total number of calculations can be reduced.

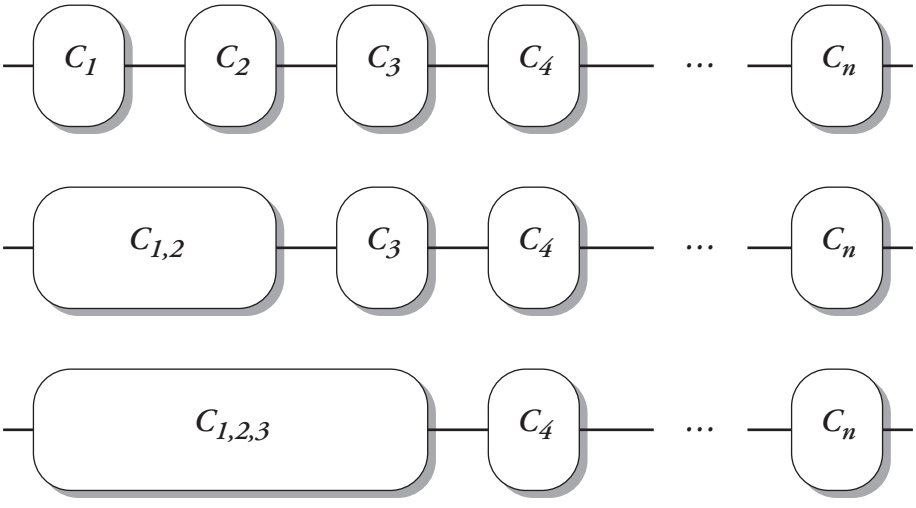


Figure 5.1 Reducing the number of states in a calculation

5.5 THE USER INTERFACE

In this section we will focus on the user interface of the proof assistant. Like the simulator, the proof assistant is an interactive program that communicates with the user by way of several windows. In this section we will describe the different windows of the proof assistant and their functionality. Because the implementation of the proof assistant is based on the implementation of the simulator, large parts of software have been reused. In the following sections we will describe the different windows that make up the user interface of the proof assistant.

5.5.1 THE PSF WINDOW

The *PSF* window displays a PSF version of the TIL specification that was used as input for the simulator. This translation from TIL to PSF does not yield the original PSF specification because TIL does not support any modular structure. However, PSF demands a division of a specification into data modules and process modules at least. Therefore, the translation consists only of one data module and one process module. All *skip* actions that occur in a specification are extended with a unique number according to the following format: *skip*<*n*>, where *n* is a number.

The text in the window may be scrolled with the use of the mouse device, according to the default scrolling possibilities for windows offered by the X Window System. The representation of the complete PSF text resides on a disk file in the file system. This has as effect that scrolling may be slow due to intensive I/O traffic for the disk involved.

5.5.2 THE VERIFY WINDOW

The *Verify* window is the main window of interaction with the user. The proof that is built up of applications of simple axioms or tactics is registered here step by step.

Initially the user has to choose a process to start the proof with. After selection, the process identifier and its defining process expression are shown in the *Verify* window. Next the user can select a sub-expression that should be rewritten with the mouse. If there are any axioms that can be applied to the selected expression, they will be shown in the *Rewrite* window.

Selection of an expression other than an atom or process identifier, is performed by selecting the outermost operator in the parse tree representation of the expression. Double clicking on a process identifier results in the expansion of this identifier into its defining process expression.

After applying one of the operations described above to an expression, the proof assistant rewrites the chosen expression and displays the complete expression in its changed state. The expression is preceded by a combination of a character and a digit that identify the proof step. This is a useful feature to retain view of the proving process, because the user can *undo* a step or can switch over to another proof. The character identifies a proof sequence. It starts with *'A'* and changes every time the user starts a new proof sequence. The digit identifies the number of the current step within the proof. It starts with *'1'* and increases at every new step in the proof. This means that the first step of a proof in a program session is labeled with *(A1)*. After the user has changed to a new process

definition, the proof assistant inserts the message: "*Other Definition*" between the two proof sequences in the *Verify* Window.

The text in the window may be scrolled with the use of the mouse device, according to the default scrolling possibilities for windows offered by the X Window System. As opposed to the text in the *PSF* window, the information in the *Verify* window resides in memory. Scrolling performance in the latter window is therefore normally better, except when the system is *'swapping'* extensively.

5.5.3 THE REWRITE WINDOW

The *Rewrite* window is only visible when the user has selected an expression in the *Verify* window. The *Rewrite* window contains the possible rewrite steps for the selected expression. The user can select one of these rewrite steps or make the window disappear by choosing the *quit* button.

5.5.4 THE OPERATIONS WINDOW

The tactics we discussed in section 5.4 can be called from the *Operations* window. Next we will give a list of the possible commands that can be issued from the *Operations* window.

- *find deadlocks*
  tries to find deadlocks by examining atomic actions within an encapsulation that cannot escape blocking.
- *eliminate deadlocks*
  removes multiple deadlocks in one step by applying axiom *DLK_ALT*.
- *head tail*
  rewrites the selected term into head normal form.
- *rec head tail*
  applies the *head tail* strategy recursively. The user has to take care that the chosen expression does not lead to an endless recursion.
- *fold term*
  searches the internal data base for a process variable defined as the currently selected process expression. If such a process variable exists the selected process expression is replaced by this process variable, otherwise the user is asked for a name and the program creates a new process variable, defined as the selected expression, and replaces the selected expression with the newly created process variable.
- *encapsulate term*
  'pushes' an encapsulation through a merge. (See section 5.4.3)
- *hide term*
  'pushes' an abstraction through a merge. (See section 5.4.3)
- *hide encaps term*
  is a combination of the two previous tactics. It *'pushes'* an abstraction through an encapsulated merge. (See section 5.4.3)

5.5.5 THE SPECIAL WINDOW

All global commands for the proof assistant can be issued from the *Special* window. The possible commands will be explained below.

- *quit*
  leaves the proof assistant and returns to the operating system.
- *process status*
  gives a dump of the internal representation of the current process expression. See section 4.5.5 for a detailed description of the format of this dump.
- *undo step*
  removes the latest proof step and returns to the previous state.
- *proof to troff*
  generates an output file containing the current proof in *troff* [Oss77] format.
- *reset*
  removes the current proof and returns to the initial state.
- *change definition*
  switches over to a different process definition during proof construction.

5.6 THE IMPLEMENTATION OF THE PROOF ASSISTANT

In this section we will briefly sketch the structure of the actual implementation of the proof assistant. For the implementation of the proof assistant large parts of the software of the simulator have been reused. Like the simulator the proof assistant was implemented in C [KR88] using the X Window System. The functional decomposition of the proof assistant is given in Figure 5.2.

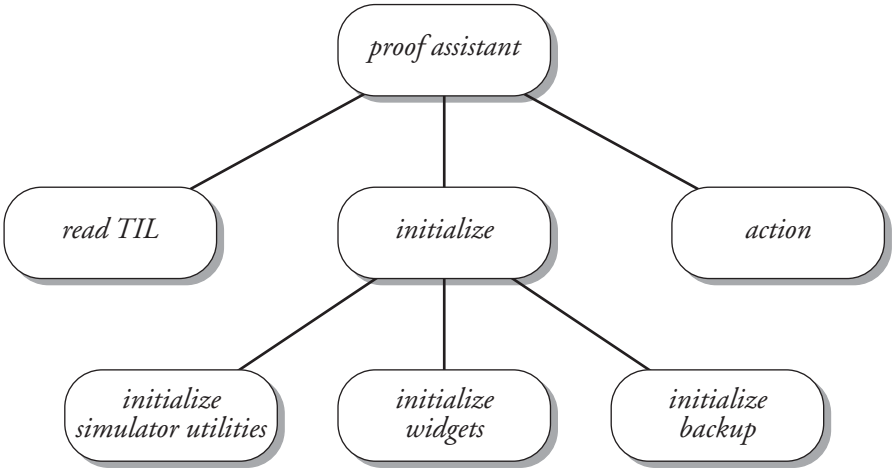


Figure 5.2 Top level functional decomposition

At the top level the simulator is decomposed into three functions. First the TIL input file is read (*readtil.c | h*, *readtiltype.c | h*, *readexpname.c | h*, *fieldex.c | h*, *prtilparts.c | h*, *tiltype.h*) then several initializations are performed and the main loop of the simulator is entered in the box labeled with *action*.

The initialization is split up into several domains:

- initialization of utilities that are literally taken from the simulator
  simutil.c | h
- initialization of X Window entities, widgets and callback routines
  widgets.c | h
- initialization of the package that takes care of backing up rewrite steps, which are needed for the *undo* function. Moreover this package contains routines to write the proof to an output file.
  steps.c | h

The functional decomposition of the main loop (*action*) of the proof assistant is given in Figure 5.3.

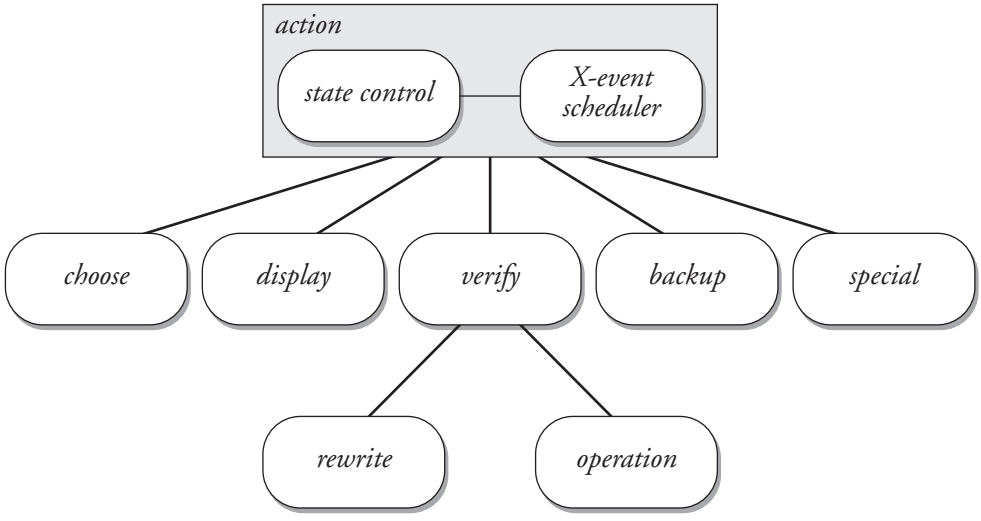


Figure 5.3 Functional decomposition of the main loop of the proof assistant

Module *state control* determines which actions are to be performed (*statecontrol.c | h*). It cooperates closely with the X-event scheduler, which is taken from the X libraries. The X-event scheduler deals with all events that have to do with the user interface such as: registering mouse clicks, resizing and moving windows, moving and changing the cursor.

Now we will describe the second-level modules. Module *choose* contains routines to allow (visual) selection of an item from a list (*choosewid.c | h*). These routines are used in selecting process expressions.

The routines that are used to convert the internal representations of process expressions into human-readable expressions are gathered in module *display* (*display.c | h*).

Module *verify* is divided into two parts. The routines for rewriting a process expression, which actually implement the ACP axioms, are in *rewrite.c* | *h*. The so-called tactics are implemented in module *operation* (*operation.c* | *h*, *rewritealg.c* | *h*).

The backup facility needed for the *undo* command is implemented in module *backup* (*steps.c* | *h*).

Finally, the routines that implement special functions, such as changing to another process definition and resetting the proof assistant, are implemented in the *special* module (*specialwid.c* | *h*).

5.7 VERIFICATION OF TWO ONE-BIT BUFFERS, AN EXAMPLE

Although an explanation of an interactive tool by means of a written text is less adequate than active hands-on experience, we will try to demonstrate the working of the proof assistant with an example. We will use the tools to construct a proof that a system of two one-bit buffers shows the same behaviour as one two-bit buffer.

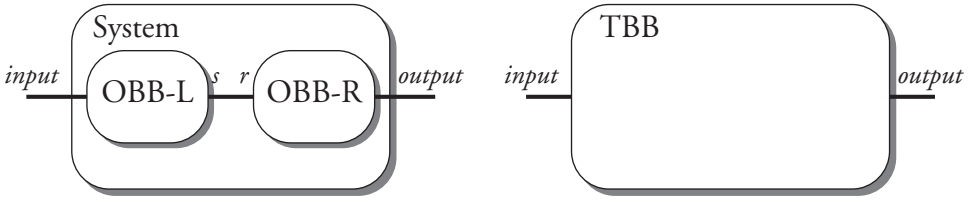


Figure 5.4 The two different buffers

5.7.1 THE SPECIFICATION

The system that we will consider consists of a parallel composition of two one-bit buffers, which are connected by an internal port. The left buffer (*OBB-L*) gets data from the environment through an *input*-port and sends it to the internal channel, while the right buffer reads data from this internal channel and hands it over to the environment through the *output*-port. The situation is as depicted in Figure 5.4.

We only consider the actions taking place at the border of the box, and we abstract from all actions at the internal port. In PSF the specification of this system looks as follows. First we define the behaviour of the two-bit buffer, which will be our target specification. The atomic actions *input* and *output* are defined, the process *TBB*, which represents the two-bit buffer, and two auxiliary processes *TBB'* and *TBB''*.

The behaviour of the buffer is straightforward. It starts with an input action and reaches state *TBB'*, which indicates that there is one item in the buffer. *TBB'* can either do another input action and continue in state *TBB''* with two buffered items, or it can do an output action and restart with an empty buffer. Process *TBB''* can only do an output action and continue with *TBB'*.

```
process module TBB
begin

  exports
    begin
      atoms
        input, output
      processes
        TBB, TBB', TBB''
    end

  definitions
    TBB   = input . TBB'
    TBB'  = input . TBB'' + output . TBB
    TBB'' = output . TBB'

end TBB
```

For the system of two one-bit buffers, we define the processes *OBB-L* and *OBB-R*. Communication through the internal channel takes place by means of the *r* and *s* action. The communication between the *r* and *s* action results in a *c* action.

The behaviour of the two one-bit buffers is defined in a straightforward way. Now the process *System* is defined as the parallel composition of these two buffers, while encapsulating unsuccessful communications (from the set *H*) and abstracting from communications occuring at the internal channel (from the set *I*).

```
process module Buffers
begin

  imports
    TBB

  atoms
    s, r, c

  processes
    System, OBB-L, OBB-R

  sets of atoms
    H = { r, s }
    I = { c }

  communications
    s | r = c

  definitions
    OBB-L = input . s . OBB-L
    OBB-R = r . output . OBB-R
    System = hide(I, encaps(H, OBB-L || OBB-R ))

end Buffers
```

5.7.2 VERIFICATION

The aim is to verify that the processes *TBB* and *System* define the same process, by which one may conclude that a composition of two one-bit buffers can be used as an implementation for a two-bit buffer. The following explanation will be interspersed with output of the proof assistant.

After starting the tool, one can select the process to be manipulated. In this example this will be the *System* process. The proof assistant starts with displaying the definition of *System*.

```
System = hide(I, encaps(H, OBB-L || OBB-R ) )
```

After clicking on the *hide* operator to select the entire expression, applying the *head-tail* operation yields expression *A1*.

```
( A1 )    = input . hide(I, encaps(H, s . OBB-L || OBB-R ) )
```

The *skip* action appears when the head-tail operation is applied to the tail of *A1*.

```
( A2 )    = input . skip . hide(I, encaps(H, OBB-L || output . OBB-R ) )
```

After selecting the first dot, an axiom can be chosen which removes internal *skip* actions in this context (*SKPACP_T1B*).

```
( A3 )    = input . hide(I, encaps(H, OBB-L || output . OBB-R ) )
```

The final step is to attach a new name, *S'* for example, to the expression after the *input* action (*A4*). This is done by selecting the appropriate expression and choosing *fold term*.

```
( A4 )    = input . S'
```

Next we focus on the newly defined process *S'*. After three steps we have been able to prove it equal to an expression which contains a new process name *S''* and the already defined process *System*. Note that the order of the left and the right buffer in the definition of *System* is opposite to the order in *B2*. However these two subexpressions are recognized by the proof assistant as being equal.

```
S' = hide(I, encaps(H, OBB-L || output . OBB-R ) )
( B1 )    = input . hide(I, encaps(H, s . OBB-L || output . OBB-R ) ) +
                output . hide(I, encaps(H, OBB-R || OBB-L ) )
( B2 )    = input . S'' + output . hide(I, encaps(H, OBB-R || OBB-L ) )
( B3 )    = input . S'' + output . System
```

The third step is to repeat the process for the new process name *S''*. This yields an expression in which the definition for *S'* is recognized automatically (*C3*) when we try to apply *fold term* to the tail part of the expression in *C3*.

```
S'' = hide(I, encaps(H, s . OBB-L || output . OBB-R ) )
( C1 )    = output . hide(I, encaps(H, OBB-R || s . OBB-L ) )
( C2 )    = output . skip . hide(I, encaps(H, output . OBB-R || OBB-L ) )
( C3 )    = output . hide(I, encaps(H, output . OBB-R || OBB-L ) )
( C4 )    = output . S'
```

The result of this manipulation is that we have given a derivation that indicates that the process *System* is the solution of the following set of equations.

```
System    = input . S'
S'        = input . S'' + output . System
S''       = output . S'
```

Now using the Recursive Specification Principle (see [BW90]) we can conclude that *System* and TBB in fact define the same process. This last step of reasoning has not been implemented in the proof assistant.

5.8 CONCLUSIONS

In this chapter we have given the description of a system that can be used to assist in the process of proving properties of process specifications. We think of the proof assistant as it is now, more as a somewhat smart electronic notebook than a full-fledged proof constructing system. It never has been our aim to be able to generate proofs automatically.

Even so we think there are still a large number of aspects on which the proof assistant can be improved. In the current version the axioms are 'hard-wired' into the code. The system would be more flexible if the user is allowed to enter a set of axioms of his own, like in the PAM System [Lin92]. In this way the user would also be able to select a different process semantics than the observational equivalence that we have implemented. To be able to achieve this, a language for representing axioms has to be developed. Moreover one can think of an extension of the PSF language that allows to express proofs, which can be checked automatically afterwards.

We think that the conventional method of state space exploration and axiomatic approach should go hand in hand. In this way the axiomatic approach can be used to cut down the state space into several components, which then can be checked by state space exploration.

Experimentation with the proof assistant has shown that the set of implemented tactics is not sufficient for anything but constructing proofs in the setting of small specifications. Several suggestions for other tactics were given, but it seemed that every new problem that was tackled asked for its own very specific extension of the proof assistant. To avoid adding many *ad hoc* extensions to the proof assistant it was obvious that more theoretical research into the area of proof theory had to performed first. The first results of this research can be found in [GP91]. Currently, the main usage of the proof assistant is in education.

CHAPTER 6
OTHER TOOLS IN THE PSF TOOLKIT

6.1 INTRODUCTION

In this chapter we will focus on some of the other tools that are available in the PSF Toolkit. These tools include a term rewriting system (*trs*), a tool that determines the initial algebra of an algebraic specification (*initial*), a bisimulation equivalence tester (*equiv*) and a tool that transforms a PSF specification into a specification in transition system format (*trans*). Moreover, we will describe the compiler driver and library manager (*psf*).

The routines in the term rewriting system are of course used by other programs in the toolkit such as the simulator and the proof assistant. The program described in this chapter is the stand-alone version that allows users to use the PSF Toolkit as an environment to study algebraic specifications without any process behaviour. In chapter 7 we will show the usage of the toolkit in this fashion in more detail by means of an elaborated example.

The *initial* and *trans* tools are merely research tools that in their current version are neither fully operational nor very user-friendly. Yet they show some interesting directions for further research.

6.2 THE PSF COMPILER

We will start this chapter with a description of the compiler driver and library manager. We recall that the PSF Toolkit deals with several (internal) languages. In the following table we list these languages as well as their default file name extensions used for files in the file system.

PSF	(Process Specification Formalism)	.psf
TIL	(Tool Interface Language)	.til
M-TIL	(Modular Tool Interface Language)	.mtil
I-TIL	(Intermediate Tool Interface Language)	.itil

The following picture shows the programs that translate a file from one language into another. The arrows show which files are transformed by which translator. The numbers at the arrows indicate how many files are involved as input or output file.

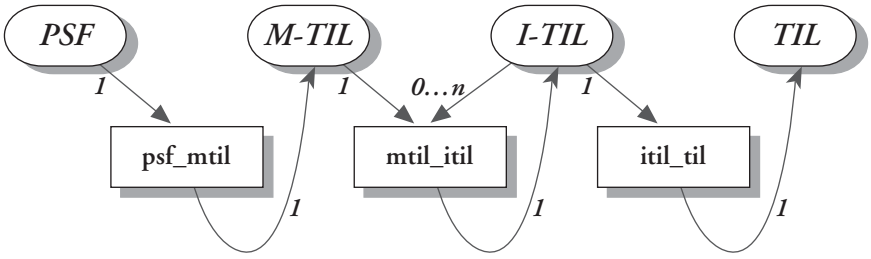


Figure 6.1 The different translators in the PSF Toolkit

The first stage in the compilation process is formed by *psf_mtil*, which contains the PSF syntax checker. This program accepts exactly one PSF module and transforms it into one M-TIL module. If a file contains more than one PSF module the file must be split on module boundaries before it is to be processed by *psf_mtil*.

The second program, *mtil_itil*, is the normalizer, which removes all modular structure from the specification. It takes as input exactly one MTIL module and zero or more I-TIL files to produce exactly one new I-TIL file.

The final stage of the compilation process is *itil_til*, which transforms exactly one I-TIL file into a TIL file.

6.2.1 PSF, the Compiler Driver and Library Manager

The user does not invoke any of the programs described above directly. The order in which the programs are called, and which arguments are supplied are governed by *psf*. The *psf* program is a combination of a compiler driver and a library manager.

The input for *psf* is a list of files containing one or more PSF modules. The *psf* program starts with determining the import structure of the specification. It creates a sorted list of modules, called the *import list*, such that, if module *A* imports module *B*, *B* occurs on the import list before *A*. The reason for this is that because module *A* depends on *B*, *B* must be compiled before *A*. If *psf* is not able to find such an ordering, execution is stopped and an error message is generated.

During the construction of the import list, *psf* starts its search for PSF modules in the input file. If any module cannot be found there, the module name is extended with *.psf* and the program starts looking for a file with this name in a set of user specified libraries. If it cannot be found there either, the program halts and reports the error to the user.

In the next step each file that contains more than one module from the *import list* is split into files containing one module each. If a file contained one module a symbolic link to this file is created. These files and links are created in a temporary directory in which later all intermediate files will also be created. The default name of the temporary directory is */tmp/psf?????* (where *?????* stands for a random five-digit integer) but can be

overridden by the user. The name of the temporary directory is recorded in a file (*.psfrc*) in the working directory.

In the second phase, *psf* starts to invoke the parser and the normalizer with different arguments for each PSF module, in the order dictated by the import list. The M-TIL and I-TIL files produced by the parser and normalizer are not discarded but stored by the library manager, so they can be used again during subsequent calls to *psf*. The final result of a compilation is a TIL file, which is stored in the directory from which *psf* was called.

The library manager keeps track of the files that have been changed since the latest compilation by inspecting the time stamps that the operating system attaches to the physical files. This way the library manager can guarantee that on successive calls to *psf*, only those files that have been edited, or files that depend on edited files, are recompiled. This means that *psf* implements *separate compilation* of PSF files.

6.2.1.1 OPTIONS

The compiler offers a large number of options to change its default behaviour.

-c compiles all files again irrespective of their age. Moreover, after successful compilation, all intermediate files are removed as well as the directory that contained these files in case it has become empty and its absolute path name contains a component *tmp*.

-i ignores any M-TIL and I-TIL files in the libraries and removes possible links to these files.

-l lists all modules that were found and the files in which they were contained in the order determined by the *import list*. The compiler exits after producing this list.

-m *modulename* defines modulename to be the target module, the module for which a TIL file is to be produced. The default target module is the last module in the *import list*.

-o *outputfile* defines the name of the output file. The default is *module.til* where *module* is the target module, unless the **-l** or **-x** option is used, in which case the default is standard output.

-s invokes *sort2set* after producing a TIL output file.

-t invokes *trs_check* after producing a TIL output file.

-v verbose mode. Prints messages on standard output, indicating progress of the compiler as well as the version number.

-x extraction mode. Copies all modules in the order determined by the *import list* to the output file specified with the **-o** option, or to standard output if no output file was specified. The compiler exits after printing all modules.

6.2.2 POST PROCESSORS

After the compiler has produced a TIL file, this file can be analyzed and transformed by two post-processors that we will describe in the following sections.

6.2.2.1 TRANSFORMING SORTS INTO SETS

The type of the placeholders used in the PSF *sum* and *merge* expressions is restricted to enumerated (finite) sets in the simulator. This restriction is reasonable because the use of an infinite data type in such an expression would lead to an infinite calculation in the simulator. To guarantee that the input to the simulator is free of infinite data types *sort2set* has been developed as a post processor.

The *sort2set* program uses a fairly crude algorithm to transform general sets and sorts into enumerated sets. All functions that have the sort to be converted as output type are considered first. If these are all constant functions, that is, they do not have arguments, an enumerated set is constructed from these functions. If a function has arguments, the sort at hand can only be converted when the output sorts of the arguments can be converted.

In the following phase sets that are constructed by the use of set operators are transformed into enumerated sets. This is of course only possible when the arguments to the set operators are enumerated sets.

The *sort2set* program has the following options.

-n rewrites the elements in the enumerated sets into normal form.

-r generates a report describing which sorts and sets are converted.

-s converts only sorts and sets that are actually used in *sums* and *merges* in the specification.

6.2.2.2 CHECKING EQUATIONS

The *trs_check* program checks some static constraints that are imposed to term rewriting systems. It checks for the following three cases:

- the left hand side is just a variable
- the right hand side contains unbound variables
- a condition contains unbound variables (at both sides)

These checks can be made in a straightforward way. The following PSF specification gives an example of the three error cases mentioned above.

```
data module Errors
begin

  sorts
    S
  functions
    a : -> S
    f : S -> S
  equations
    [ERROR1]  x = a
    [ERROR2]  f(x) = y
    [ERROR3]  f(x) = a  when  z = y

end Errors
```

6.2.3 CREDITS

The current implementation of the compiler driver and library manager is a C program written by J.C. Mulder. The *sort2set* post processor was implemented by B. Diertens and *trs_check* by G.J. Veltink.

6.3 TERM REWRITING

The evaluation of all data terms in the PSF Toolkit is implemented using *term rewriting*. In this section we will give a brief description of the basics of term rewriting. For more detailed information about term rewriting we refer to the literature. The theory of term rewriting systems is described in [DJ90,Klo91], for example, whereas [BW89] discusses the implementation of algebraic specifications in general.

A term rewriting system in the context of the PSF Toolkit consists of a *signature* and a set of *rewrite rules* or *equations*. The signature consists of a set of *sorts*, a set of *functions* and a set of *variables*. Each function has an *input type* and an *output type*. The output type is a sort, whereas the input type is a, possibly empty, list of sorts. Functions with an empty input type are called *constants*.

Using the elements from the signature it is possible to define the set of *terms* over a signature inductively:

- All variables and constants are terms.
- If f is a function with input type $s_1,\ldots,s_n$ ($n \geq 1$) and $t_1,\ldots,t_n$ are terms for which the output type of t_i equals s_i, then $f(t_1,\ldots,t_n)$ is a term.

A rewrite rule consists of a pair of terms and is written: $l = r$. There are two restrictions that apply to rewrite rules:

- The left-hand side l is not a variable.
- All variables occurring in the right-hand side r are contained in l.

The process of term rewriting is a sequence of transformations of a term. The possible transformations are dictated by the set of rewrite rules. The first step in rewriting a term t, consists of finding a sub-term s of t that *matches* the left-hand side of one the rewrite rules. Such a sub-term s is called a *redex* of t. As a result of matching, possible variables occurring in the rewrite rule are bound to sub-terms of s.

In the second step of rewriting, the occurrence of s in t is replaced by the right-hand side of the matching rewrite rule, thereby substituting possible variables in the right-hand side with the sub-terms of s they were bound to, to form a new term t'. This process is repeated for the new term t' until no more matching left-hand sides can be found. The resulting term is said to be in *normal form*.

The type of term rewriting system described above is called a *many-sorted* term rewriting system. In fact the term rewriting systems used in the PSF Toolkit are *conditional many-sorted* term rewriting systems. The conditional part refers to *conditions* that can be added to

a rewrite rule, meaning that a rewrite rule can be applied if and only if the conditions hold. A condition consists of a pair of terms and is written: $l = r$. A condition holds when both terms l and r have the same normal form. A set of conditions holds when all conditions in the set hold.

6.3.1 Reduction Strategies

A term can contain more than one redex. The order in which these redexes are rewritten induces a so-called *rewriting strategy*. One important difference between rewriting strategies is the place where the redexes are looked for. Some strategies look for redexes as deep as possible in the tree representation of a term, and so are called *innermost* strategies. *Outermost* strategies, on the other hand, look for redexes on the highest level within the term.

Within these two classes there is another division. One possibility is to try to find as many redexes as possible according to an innermost or outermost strategy and then rewrite them in *parallel*. The other possibility is to try to find just one redex, rewrite it and restart the search for redexes in the newly created term. Depending on the location where this redex is looked for we speak of *leftmost* and *rightmost* strategies. There is one more strategy called the *full substitution* strategy, in which all redexes are rewritten simultaneously.

The performance of a certain rewriting strategy depends on the type of equational specification. The leftmost-innermost strategy, for example, can only find normal forms in *strongly normalizing* specifications, while on the other hand the parallel outermost strategy is also able to find normal forms of *weakly normalizing* specifications.

There is also a difference in efficiency between the rewriting strategies. The innermost strategies are in general more efficient than the outermost strategies, because the former can look for redexes in a smaller part of the term, thereby reducing the time spent parsing sub-terms and the amount of internal administration during traversal of the tree. This also implies that an innermost strategy is more efficient with regard to memory usage.

6.3.2 An Example of Rewriting

We will illustrate the technique of rewriting using the different rewriting strategies by means of an example in which we will use the following specification of the *booleans*.

```
data module Booleans
begin

  sorts
    BOOLEAN

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

  variables
    x : -> BOOLEAN
```

```
    equations
    [01] not(true) = false
    [02] not(false) = true
    [03] and(true, x) = x
    [04] and(false, x) = false
    [05] or(true, x) = true
    [06] or(false, x) = x

end Booleans
```

In the following figures we will illustrate the process of rewriting the term *and*(*true*,*or*(*not*(*false*),*not*(*true*))) for each of the five strategies. The grey area in a term indicates the redex that is rewritten. These figures are inspired by the figures from [BW89].

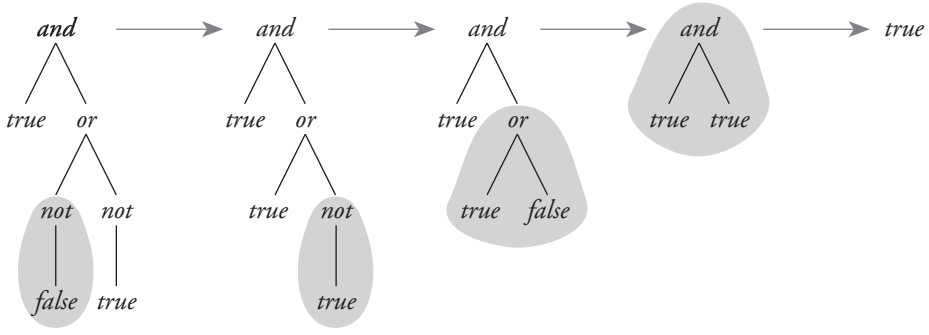


Figure 6.2 The leftmost-innermost reduction strategy

The leftmost-innermost strategy rewrites the leftmost redex that does not contain another redex.

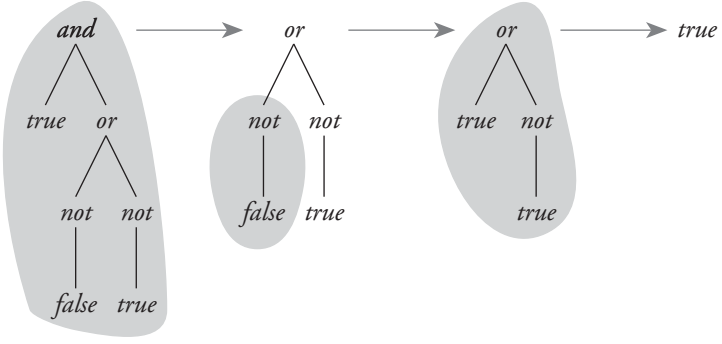


Figure 6.3 The leftmost-outermost reduction strategy

The leftmost-outermost strategy rewrites the leftmost redex that is not contained in another redex.

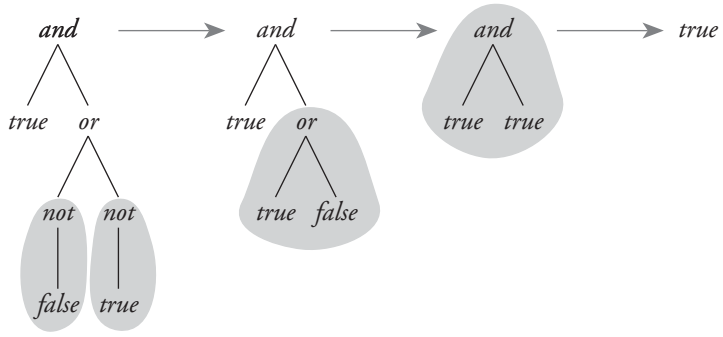


Figure 6.4 The parallel-innermost reduction strategy

The parallel-innermost strategy rewrites all redexes that do not contain another redex simultaneously.

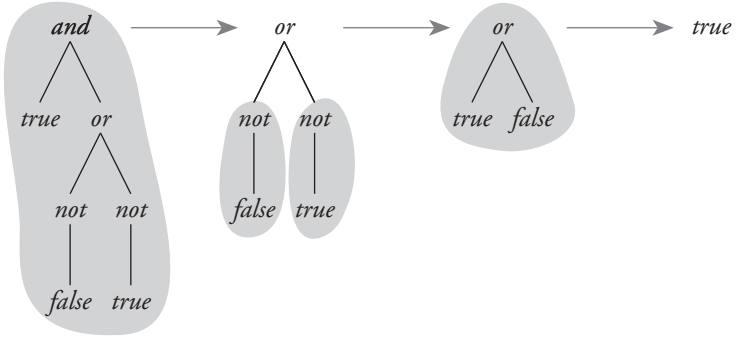


Figure 6.5 The parallel-outermost reduction strategy

The parallel-outermost strategy rewrites all redexes that are not contained in another redex simultaneously.

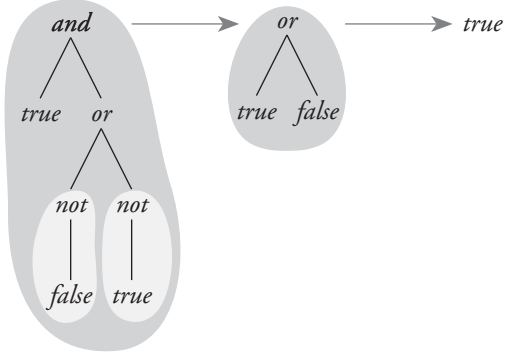


Figure 6.6 The full substitution reduction strategy

The full substitution strategy rewrites all redexes simultaneously.

6.3.3 THE TERM REWRITER

The term rewriter in the PSF Toolkit is a standard term rewriting program supporting *conditional equations*. The kernel of the term rewriter is used by other tools in the PSF Toolkit. However, the term rewriter can also be used as a stand-alone program.

The PSF language and the Toolkit can be used as just an algebraic specification environment. By specifying only data modules one can treat the PSF Toolkit as an implementation of an algebraic specification language. The PSF Toolkit is used as such for several years now, in courses on Algebraic Specification Methods at the University of Amsterdam.

The term rewriter takes as input a TIL file that specifies the sorts, functions and equations used in the specification. The terms to be rewritten are given in a second input file or are supplied interactively.

In the following sections we will describe the main routines in the term rewriter by means of pseudo-code.

6.3.3.1 TYPING OF THE INPUT TERM

If the term rewriter is used as a stand-alone program, the input term supplied by the user must be typed before it can be processed. In case other tools call the term rewriter this is of no concern, because within the PSF Toolkit it is guaranteed that tools exchange well-typed terms only. The term rewriter uses *inside-out typing* to determine the internal representation of an input term.

The syntax used for the input terms is equal to the syntax for terms in PSF. When there are items with the same name and types for their arguments, they must be disambiguated by supplying the module of origin using the following syntax: *Module-Name.Item-Name*.

When during the process of typing an error is found, the term rewriter generates an error message, followed by the input term, in which the string '*???*' is inserted after the position where the error was found. In the case that infix operators are used the string '*???*' may be placed a little further then the location where the error was found.

6.3.3.2 REWRITING A TERM

The main function is *rewrite_term*, which takes as argument a term *t*. Because rewriting is performed according to an innermost strategy, the first step is to rewrite all children of *t*, with recursive calls to *rewrite_term*. After this, it is guaranteed that all the children of term *t* are in normal form.

Next, we try to find an equation *eq* of which the left-hand side matches term *t*. If such an equation cannot be found *t* is in normal form and *rewrite_term* returns this normal form. If a matching equation can be found, *t* is replaced by the right-hand side of *eq*, the variables that were bound during the matching process are substituted and finally *rewrite_term* is called with the new *t* as argument.

```
rewrite_term(t)

  for all children i of t do
    rewrite_term(child i of t)
  od
  eq := matching-equation(t)
  if (eq = no_match) {
    return(t)
```

```
    else
     t := eq.rhs
     substitute bound variables in t
     return(rewrite_term(t))
   fi
```

6.3.3.3 FINDING A MATCHING EQUATION

The function *matching_equation* takes as argument a term *t*. Its task is to find an equation that can be used to rewrite *t*. This is achieved by scanning the list of equations until one of the left-hand sides of the equations matches the argument term *t*. If the matching equation *eq* has no conditions it is returned immediately. Otherwise, the function *conditions_valid* is called to verify whether the conditions for *eq* hold. If no matching equation is found the function returns *no_match*.

```
matching-equation(t)

  for all equations eq do
    if (match_terms(t,eq.lhs) = match) then
      if eq has conditions then
        if (conditions_valid (eq)) then
         return(eq)
        fi
       else
        return(eq)
      fi
    fi
  od
  return(no_match)
```

6.3.3.4 CHECKING CONDITIONS

The function *conditions_valid* takes as argument an equation *eq*. Its task is to check whether the conditions for *eq* are valid. The conditions are evaluated in the order in which they appear in the specification. The first step is to create two terms *t1* and *t2* from the left-hand side and right-hand side of the condition *cond* in which all variables bound in matching the left-hand side of the equation or previous conditions are substituted. If both *t1* and *t2* are *open* terms, that is, they still contain unbound variables, an error message is generated. Otherwise, the function *match_terms* is called to check that *t1* and *t2* can be matched. The first argument of *match_terms* must be a *closed* term so *t1* and *t2* are swapped if necessary. If the match fails the conditions did not hold and *false* is returned, otherwise the next condition is tested or if the current condition was the last one, *true* is returned.

```
conditions_valid(eq)

  for all conditions cond of eq do
    t1 = substitute_bound_vars(cond.lhs)
    t2 = substitute_bound_vars(cond.rhs)

    if both t1 and t2 are open terms then
      report "unbound variables in condition"
      return(false)
     else
      if t1 is an open term then
```

```
        swap t1 and t2
      fi
      if (match_terms(t1,t2) = no_match) then
        return(false)
      fi
    fi
  od
  return(true)
```

6.3.3.5 MATCHING TWO TERMS

The heart of the matching process is function *match_terms*. For reasons of simplicity we will describe a version of *match_terms* in which the first argument must be a closed term. The first test determines whether the top symbol of *t2* is a function or a variable.

In case of a function it is checked whether the indexes agree, and only then all children of *t1* and *t2* are matched with each other by a recursive call to *match_terms*. If all children can be matched the two terms are equal.

If *t2* is a variable there are two cases: either the variable is bound or free. If the variable is already bound *match_terms* is called with the same *t1* as first argument but with the value of *t2* as second argument. Otherwise, *t1* is bound to the variable represented by *t2* and *match_terms* returns *match*.

```
match_terms(t1,t2)

  if (is_function(t2)) then
    if (t1.index <> t2.index) then
      return(no_match)
     else
      forall children i do
        if (match_terms(t1.child[i],t2.child[i]) = no_match) then
          return(no_match)
        fi
      od
      return(match)
    fi
   else    // t2 is a variable
    if (t2 is bound) then
      return(match_terms(t1,value(t2)))
     else
      bind t1 to the variable represented by t2
      return(match)
    fi
  fi
```

6.3.3.6 TRACING

The term rewriter offers a *trace* option to let the user inspect which rewrite rules are applied and what the intermediate results are. Tracing is activated by the command: > *trace on*, which must be part of the input file. Tracing is deactivated by: > *trace off*. The '>' symbol is part of the command, and is used to distinguish commands from input terms.

Below we will give an example of a *trace* produced by the term rewriter. In this case a term is reduced using the specification of the booleans from the previous section.

```
and(true,or(not(false),not(true)))      =
   not(true)
   -> false
   not(false)
   -> true
   or(true,false)
   -> true
   and(true,true)
   -> true
true
```

The current implementation of the term rewriter uses a *rightmost-innermost* rewriting strategy, which can be deduced from the trace of rewrites.

6.3.4 OPTIONS

The following two options can be used to influence the behaviour of *trs*.

-r reverses the order in which the list of equations is searched for a matching equation.

-v allows for variables to occur in input terms.

The term rewriter offers an option to reverse the order in which the list of equations is searched for a matching equation. In a *confluent* specification it makes no difference which equation is chosen from a set of possible equations. However, it is a common error to incorporate order dependencies in specifications, especially among inexperienced writers of algebraic specifications. By comparing the results of the term rewriter using both normal and reversed order, non-confluent systems can be detected more easily.

6.3.5 POSSIBLE EXTENSIONS

There are several extensions to the term rewriter we could think of. The most obvious extension would be the ability to choose from different rewriting strategies. It is sometimes the case that specifications can be given in a more natural way by a weakly normalizing system, than by a strongly normalizing system. However, in case of a weakly normalizing system, the current rightmost innermost strategy will almost certainly get into an infinite branch of rewrites. When there would be the possibility to use an outermost strategy in such a case, the term rewriter could be used for a larger class of algebraic specifications.

A second extension would be the detection of cycles in rewriting. The current implementation of the term rewriter cannot recognize infinite rewrites. If there is an infinite recursion the program will rewrite until all stack space has been used, which will result in an abort by the operating system software. Cycles in the rewriting process can be detected by storing redexes on a stack as they are rewritten. Each time a new redex *r* is to be rewritten the term rewriter can check the stack of redexes for an occurrence of *r*. If *r* induces its own rewriting there is clearly a cycle and the term rewriter can halt and give a warning indicating the cycle.

A third extension would require information about the constructor functions of a data type. Constructor functions are those functions that are used in constructing normal forms.

The term rewriter can warn the user as soon as a normal form containing a non-constructor function appears, because this situation normally indicates some kind of error. Information about constructor functions is currently not incorporated in PSF specifications. A rough approximation could be to interpret all functions for which there are no equations as constructor functions.

6.3.6 CREDITS

The term rewriter in the PSF Toolkit is a C program written by C.H.S. Dik using techniques described in [Dik89] and [Kap87]. The current version of the term rewriter has replaced the Prolog based implementation of the ASF project. The latter program was mainly written by P.R.H. Hendriks.

6.4 CALCULATING INITIAL ALGEBRAS

When we write down an algebraic specification, we usually have a specific model in mind that we try to describe. However, there is a fair chance that we have made a mistake somewhere in the specification. This is particularly so for specifications of large and complex systems. Most errors result in an initial algebra of the specification at hand that does not comply with the model we have in mind.

There have been attempts to restrict the expressiveness of an algebraic specification language in such a way that the initial algebra of a specification can be determined easily. One of these attempts is *Perspect* [Wie91] in which one is forced to write *persistent* specifications. Experiments carried out with Perspect showed that the restrictions imposed by the language influence the specification style significantly and are to be regarded as too restrictive for a large class of specifications.

In the PSF Toolkit we have taken a different approach. The language allows for a larger class of specifications. Tracking down errors in the specification, with respect to the initial algebra, is facilitated by the use of a tool that generates the initial algebras of the sorts defined in a given specification of data types. The initial algebra generator from the PSF Toolkit, which is called *initial*, tries to calculate a finite projection of the initial algebra for the sorts specified in its TIL input file, by enumerating normal forms. The calculation yields only a finite projection because a large number of specifications incorporate infinite sorts, so that calculating their initial algebras would take infinite time. The finiteness of the calculation is guaranteed by specifying an upper bound for the number of elements that an initial algebra may contain. If the program passes this limit it stops generating more normal forms for this sort, and reports this to the user.

In this section we will describe the implementation of the initial algebra report generator and give an example of its usage.

6.4.1 IMPLEMENTATION

As described above, *initial* generates a finite projection of the initial algebra of the sorts in a PSF specification. We should stress that the tool only operates correctly for *strongly normalizing* term rewriting systems. This is due to the fact that it simply enumerates all possible normal forms in a systematic way. If a term rewriting system is not strongly normalizing *initial* could possibly get into an infinite rewriting sequence and never return an answer.

The program starts with determining the dependency relation between the sorts in the given specification. A sort S is said to be *depending* on T if there is a function f with output type S, in which T is part of the input type.

$$f\colon T_1 \# \ldots \# T_n \rightarrow S \qquad \Rightarrow \qquad S \text{ depends on } T_i \qquad (1 \leq i \leq n)$$

In the next step the program creates a list for each sort. These lists will be used to register the sort's normal forms.

Elements of the initial algebra are said to be found in so-called *segments*. Segment *0* is formed by the normal forms of all constant functions. These normal forms are added to the appropriate normal form lists. During the entire process of generating the initial algebras these lists contain unique normal forms. This implicates that duplicate normal forms are discarded.

In the next phase the program starts generating normal forms in an inductive way. The program inspects all non-constant functions in the specification. For each function and for all possible combinations of arguments, using the terms on the normal form lists as arguments, it constructs a term and rewrites this term into its normal form. Then it adds this normal form to the appropriate normal list if it is not already on the list. Figure 6.7 gives a graphical impression of this process.

To prevent unnecessary duplicate calculations it is required that in determining a normal form in segment n at least one of the arguments is a normal form from segment n-1.

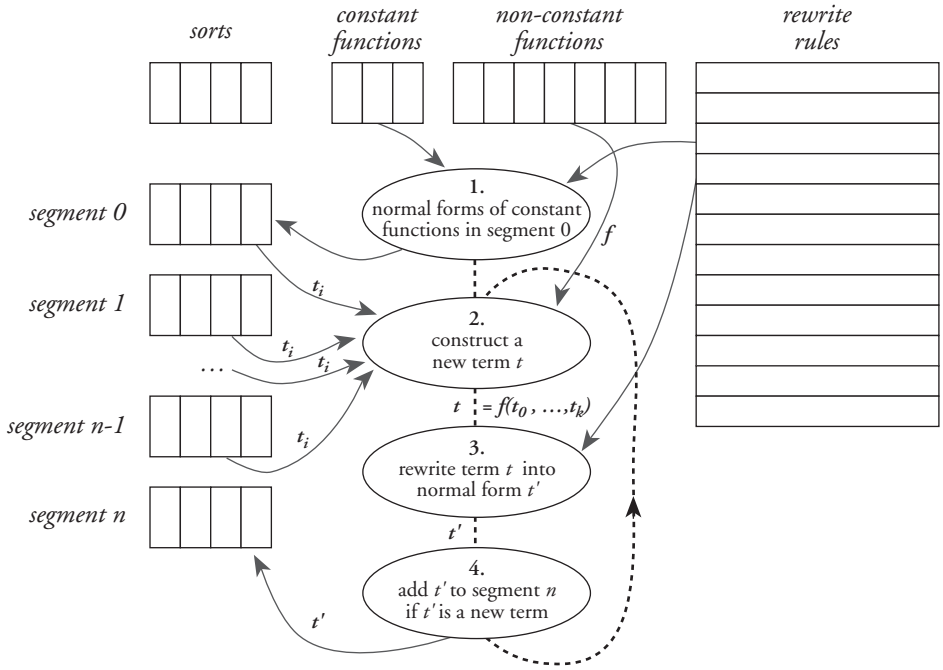


Figure 6.7 Graphical representation of the *initial* implementation

This phase is repeated, each time generating a segment on a higher level, until no new combinations of arguments can be found. This is detected by means of the notion of *activity* of a sort.

A sort is said to be *active* in segment *n*, if a sort on which it depends was extended with new normal forms in segment *n-1*. If there are no more active sorts, the calculation halts and the number of normal forms found for each sort are reported to the user.

6.4.2 OPTIONS

Because *initial* is mainly used as a tool to find errors in a specification, it contains a number of options that let the user influence its behaviour to be able to track the problems in an optimal way. The options are listed below in alphabetical order:

-c can be used to show the contents, the normal forms, of each initial algebra.

-d shows the dependencies between sorts.

-f shows the name of the function currently examined for rewriting as well as the number of combinations of its arguments. If the program seems to be idling for a while this option can be used to show where it is spending so much time.

-g shows each new element of an initial algebra as soon as it is encountered.

-o uses only constructor functions to generate new normal forms. Constructor functions in this case are defined as functions that have no accompanying rewrite rules. **Caution**: Usage of this option may result in incomplete initial algebras.

-r shows all rewrites that are performed in order to find new normal forms.

-s *size* gives the maximum number of elements of the initial algebra for any sort. If this limit is exceeded for a certain sort the program does not try to find more elements of its initial algebra. The default *size* is 100.

-v starts the verbose mode. The version number of initial is shown on startup as well as the numbers of the segments as soon as they are entered.

6.4.3 EXAMPLE

An example of the output of *initial* generated for the *integers* from the PSF Standard Library is given below. The default value (100) for the maximal size of the initial algebras is used in this example and the output has been edited, to cut down its size.

```
*** Initial Algebra Report ***

'INTEGER': 101 element(s)
*** WARNING : possibly infinite sort. Initial Algebra size reached limit
- (^ 0)
- (^ 1)
- (^ 2)
  ...
- ((^ 9) ^ 9)
- (((^ 1) ^ 0) ^ 0)
```

```
'NATURAL': 101 element(s)
*** WARNING : possibly infinite sort. Initial Algebra size reached limit
- zero
- s(zero)
- s(s(zero))
- s(s(s(zero)))
  ...

'BOOLEAN': 2 element(s)
*** WARNING : (in)directly based on possibly infinite sort
- true
- false

'DIGIT': 10 element(s)
*** WARNING : (in)directly based on possibly infinite sort
- 0
- 1
  ...
- 8
- 9
```

The warning for the sort *BOOLEAN* is the result of functions with (possibly) infinite data domains that have *BOOLEAN* as output type. An example of such a function is the equality function for *Naturals*.

6.4.4 CREDITS

The *initial* program in the PSF Toolkit is a C program written by G.J. Veltink.

6.5 GENERATING TRANSITION SYSTEMS

The algorithm implemented in the bisimulation equivalence tester *equiv* demands that the specifications to be analyzed are given in the form of a state transition system. A state transition system is described by a set of possible states of a system and a set of, possibly labelled, transitions between states.

A general PSF specification does not comply with this format. However, a subset of PSF specifications can be considered as state transition diagrams, namely those specifications that solely consist of process definitions in the following format:

$$X = a \,.\, Y$$

In this format X and Y define states and a is a label. The complete expression indicates that from state X one can do an a and reach state Y. In this section we will describe an experimental tool that transforms a general PSF specification into a PSF specification in the state transition format.

For LOTOS, there exists a tool with similar functionality called Cæsar [GS90]. This tool translates specifications written in a LOTOS subset first into extended Petri Nets, then into state graphs. We will follow a different approach in our implementation of the transition system generator.

6.5.1 RESTRICTIONS

The algorithm we are going to present imposes some restrictions on PSF specifications it can handle. The main restriction is that the specifications cannot contain any data types. We agree that this is a serious restriction, but we have decided to study the problem without data first, in order to get a certain feeling for the problem at hand.

As a result of this restriction, the specifications to be considered consist of a finite number of process definitions. If we would allow specifications to contain data types, it would be possible to give specifications consisting of an infinite number of equations. An example of such a specification, where variable *i* is an integer, is:

$X(i) = a(i) \;.\; X(i+1)$

The format we are to use to represent transition systems is only capable of representing finite systems, so any attempt to catch the above-mentioned specification in this format is pointless.

6.5.2 GENERATING TRANSITIONS

The basic idea behind the algorithm described in the following sections is that every process expression encodes a certain system state. The expression that remains after execution of an atomic action encodes the state that can be reached by a transition labelled with this action. We will illustrate this concept by means of an example using the following simple recursive specification:

$X = a \;.\; b \;.\; X$

The corresponding transition system is given in the following figure:

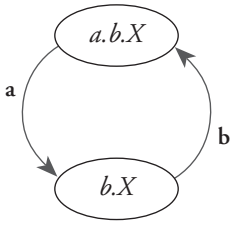


Figure 6.8 A simple transition system

The key of the algorithm is to find a systematic way to list all process expressions resulting from a specification and the transitions between them. To determine equality between two states we will assume syntactical equality for the time being. This implies that the expressions $a + b$ and $b + a$ are treated as different although they are equivalent within the semantics of PSF. This implies that the generated transition systems are not necessary *minimal*. However, this anomaly does not influence the result of *equiv* and because the main target for now is to generate a transition system we will ignore it.

In the following we will present the algorithm that constructs the transition system. The result will be two sets: *S* the set of states consisting of process expressions and *T* the set of transitions consisting of triples of the form <*state* # *label* # *state*>.

A label in this tuple can be any atomic action, the constant process expression τ representing *skip* the internal action or ε the *empty label*. The empty label indicates that

one can reach one state from another without executing any atomic action. This ε is related to the ε-move as known in the field of finite automata [HU79] and should not be confused with the empty process ε as introduced in [KV85].

There are two special states that represent the so-called final states: δ represents the state of unsuccessful termination, and √ represents successful termination of a process. In the following a represents any atomic action and X and Y represent process expressions. The actual algorithm is given in the form of pseudo-code below:

step 1 ***Initialization***
$T := \emptyset$
$S := \{ \delta, \surd \}$
P := ∀ process names p from the specification

step 2 ***Test for end condition***
$P = \emptyset$ → goto step 8

step 3 ***Choose the next term t from P***
$P := P \setminus \{t\}$

step 4 ***Test if this term t represents a new state***
$t \in S$ → goto step 2
$t \notin S$ → $S := S \cup \{t\}$
$T := T \cup \{ <t, \varepsilon, t> \}$

step 5 ***Determine head normal form of the term t***
$t' :=$ hnf(t)

step 6 ***Add transitions according to the form of its head normal form (t')***
$t' = \delta$ → $T := T \cup \{ <t, \varepsilon, \delta> \}$
$t' = \tau$ → $T := T \cup \{ <t, \tau, \surd> \}$
$t' = a$ → $T := T \cup \{ <t, a, \surd> \}$
$t' = a.X$ → $T := T \cup \{ <t, a, X> \}$
$P := P \cup \{ X \}$
$t' = X + Y$ → $T := T \cup \{ <t, \varepsilon, X>, <t, \varepsilon, Y> \}$
$P := P \cup \{ X, Y \}$

step 7 ***Loop***
goto step 2

step 8 ***Termination***
halt

As an example we apply the algorithm to the following recursive specification:

$Z = X + Y$
$X = a \,.\, Y + c$
$Y = b \,.\, X$

The result is depicted in figure 6.9. For reasons of clarity we have introduced empty transitions between process names and their definitions although strictly speaking the *hnf*

function skips this intermediate step. Moreover, we have deleted the empty transitions from each state to itself.

Evidently this algorithm will not halt for the general class of PSF specifications. An example of a specification that will cause an infinite computation, although it has a finite transition system is:

$X = a \,.\, X \,.\, X$

If we would try to apply the algorithm to X, it would generate an infinite series of *'new'* states (a.X.X, a.X.X.X, a.X.X.X.X, …). We will discuss this problem later.

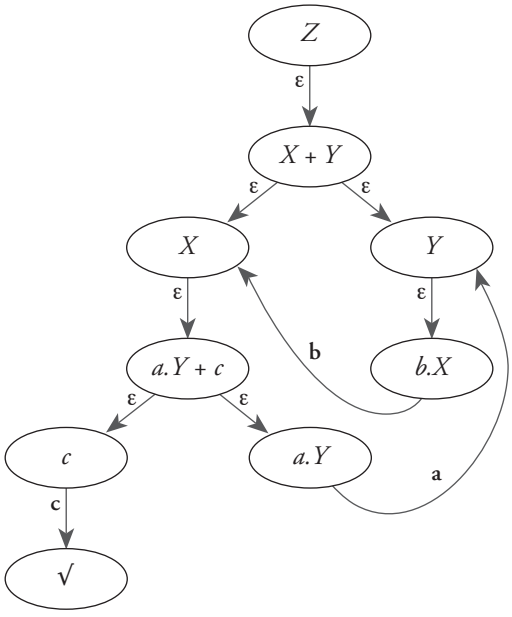


Figure 6.9 Generated transition system containing ε-transitions

6.5.3 REMOVING EMPTY TRANSITIONS

The result of the previous step is a transition system with empty transitions. We recall that a transition represents the execution of an atomic action. Therefore, the empty transitions have to be removed from the generated transition system. This process is performed by an algorithm described in [AU77]. This algorithm uses a function $E()$, the *epsilon closure*, that assigns to each state s a set of states following two rules:

- each state s' in $E(s)$ can be reached from s through a path of empty transitions. (a state can reach itself through an empty transition by default)
- there is at least one *non*-empty transition originating from s'.

Next, using this *epsilon closure*, a new transition system is calculated in which all empty transitions have been removed. To this end two sets S' and T' are introduced. Set S' will contain the states of the new ε-less transition system, consisting of epsilon closures of states from S, whereas T' will contain the new transitions between these epsilon closures. The algorithm uses the output sets S and T from the previous algorithm as input and is again given using some form of pseudo-code:

step 1 ***Initialization***
$S' := \emptyset$
$T' := \emptyset$

step 2 ***Test loop condition***
$S = \emptyset \quad \rightarrow$ goto step 6

step 3 ***Choose the next state s from S***
$S := S \setminus \{s\}$

step 4 ***Add the epsilon closure of s to S'***
$S' := S' \cup \{E(s)\}$

step 5 ***Loop***
goto step 2

step 6 ***Test loop condition***
$S' = \emptyset \quad \rightarrow$ goto step 12

step 7 ***Choose the next epsilon closure s' from S'***
$S' := S' \setminus \{s'\}$

step 8 ***Test loop condition***
$s' = \emptyset \quad \rightarrow$ goto step 6

step 9 ***Choose the next state s from epsilon closure s'***
$s' := s' \setminus \{s\}$

step 10 ***Add new transitions to T' for all transitions in T starting in s***
$\langle s,a,x\rangle \in T \wedge a \neq \varepsilon \quad \rightarrow \quad T' := T' \cup \{ \langle s',a,E(x)\rangle \}$

step 11 ***Loop***
goto step 8

step 12 ***Termination***
halt

Set T' contains the new transition system from which all empty transitions have been removed. Calculating the epsilon closures from our example leads to the following table:

s	E(s)
Z	{ b.X, a.Y, c }
X + Y	{ b.X, a.Y, c }
X	{ c, a.Y }
Y	{ b.X }
b.X	{ b.X }
a.Y + c	{ c, a.Y }
c	{ c }
a.Y	{ a.Y }
√	{ √ }

The new transition system is given in the following figure:

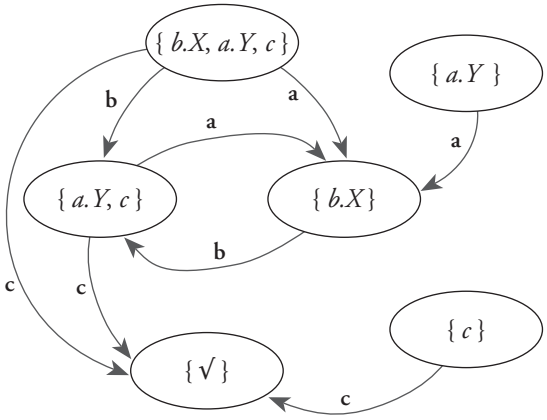


Figure 6.10 Transition system after removing empty transitions

6.5.4 MINIMIZING THE INITIAL TRANSITION SYSTEM

As mentioned earlier, the algorithm that generates the initial transition system has some drawbacks. One problem stems from the fact that it uses syntactical equivalence to test whether two process expressions are equal. However, there are more expressions equivalent according to the process semantics of PSF. To recall the example we gave earlier: *a* + *b* and *b* + *a* are equal although the algorithm does not acknowledge this fact.

To overcome this problem it is possible to define a syntactical normal form. To be able to define such a normal form we have to assume an ordering to be defined on process terms. Assuming several transformation rules such as:

x + y -> x + y when ord(x) ≤ ord(y)
x + y -> y + x when ord(x) > ord(y)

it is possible to generate syntactical normal forms. If in every stage of the transition system generation all terms are reduced using this scheme, duplicate calculations can be avoided and the size of the generated transition system is reduced.

6.5.5 HIGH-LEVEL PSF TRANSFORMATIONS

There is a class of PSF specifications that causes the algorithm that generates the initial transition system to get into an infinite calculation, although the specifications can be represented by a finite state transition system. An example of such a specification is:

$X = a \,.\, X \,.\, X$

In fact this specification could as well be written as:

$X = a \,.\, X$

for which the given algorithm is able to generate a finite transition system. The key observation is that any process expression $X.Y$ can be replaced by X when X has no option to terminate successfully.

This leaves us with the task to give a description of an algorithm that determines whether a given expression has a successful termination option. A start with the development of such an algorithm has been made in [Cro91]. However, the algorithm described there still has some minor errors.

Even if this algorithm were corrected, there are still some specifications that will be rejected being considered to generate infinite transition systems although they (also) have a finite representation. An example of such a specification is:

$X = a \,.\, Y \,.\, X$
$Y = a \,.\, Y + a$

This specification, of course, generates the same transition system as:

$X = a \,.\, X$

6.5.6 CREDITS

The transition system generator in the PSF Toolkit is a C program written by G.J. Veltink.

6.6 CHECKING PROCESS EQUIVALENCE

In this section we will focus on the final tool in the PSF Toolkit: the equivalence tester *equiv*. Equivalence testers are used to determine whether the behaviour of two processes is equal. An important example of the use of equivalence testers is checking that two specifications on different levels of abstraction are equivalent.

This technique is often used when a system is to be developed in a *top-down* fashion. First a rough specification is given, which describes the desired external behaviour of the system. In the second phase a more detailed, *implementation-oriented*, specification of the system is given and it is checked whether, after abstraction of internal actions in the detailed specification, the two systems are equivalent.

6.6.1 THEORETICAL BACKGROUND

If two processes are equivalent, there exists a so-called *bisimulation* relation between the processes. There are several types of bisimulation. The first, called *strong bisimulation*, was introduced in [Par81]. This bisimulation is very restrictive and of less practical importance in a tooling environment than *observation congruence* [Mil80], sometimes also

referred to as *weak bisimulation*. A third bisimulation, maintaining the branching structure of a process was introduced in [GW89] and is accordingly called *branching bisimulation*.

Performance is an issue for a tool environment and therefore the complexity of the algorithms determining a bisimulation is important. In this respect *branching bisimulation* has an advantage over *observational equivalence*. The latter equivalence can be decided in $O(|states|^3)$ time, whereas it is shown in [GV90] that branching bisimulation can be decided in $O(|states| . (|transitions| + |states|))$ time. This is the main reason that the *equiv* tool uses branching bisimulation as semantics.

Traditional equivalence testers are only capable of determining the equivalence of two process expressions. However, they are not able to point out why two processes are different. One solution to this problem is to generate formulae in the Hennessy-Milner Logic [HM85] that distinguish the two processes. The *equiv* program uses an algorithm to generate distinguishing formulae as described in [Kor92], which in turn is based on research described in [Cle90] and [Hil87].

6.6.2 THE IMPLEMENTATION

The *equiv* tool, the implementation as well as the theory, is extensively described in [Pol92]. In this section we will only sketch the behaviour of the equivalence tester. The *equiv* tool takes as input a TIL file in transition system format as described in the section on *trans*. This means that *equiv* too does not allow for data to be used in the specification.

The algorithm that is the basis for determining the equivalence of two processes is a partitioning algorithm. This means that all states of the transition system present in the TIL input file are initially grouped into one set called a *block*. Next, this block is split into smaller blocks which in turn are split until, in the final partition, the remaining blocks are exactly the equivalence classes of the bisimulation equivalence. In this final partition two states are bisimilar if and only if they are in the same block. After the final partition has been computed the equivalence of two processes can be determined by checking whether their initial states bisimulate.

To visualise this process we will use the example given in [Pol92]. Figure 6.11 shows the operation of the partitioning algorithm for the following two processes:

$\tau.a + b$
$\tau.a + a + b$

During the partitioning process, *equiv* keeps information about the reason why a block was split. Using this information it is possible to generate a distinguishing formula explaining why two states are not equivalent. The two processes in our example are not equivalent and the distinguishing formula is:

$$\Delta(\{s_1\},\{s_5\}) = \neg((tt\langle b\rangle tt)\langle a\rangle tt)$$

This formula should be interpreted as follows: s_1 and s_5 are not branching bisimilar because from s_5 it is possible to do an a-transition ($s_5 \rightarrow s_7$) for which it is possible to do a b-transition in all intermediate states (only s_5 in this case). Although it is eventually possible to do a a-transition from state s_1 ($s_2 \rightarrow s_4$), it is not possible to do a b-transition from the intermediate state s_2.

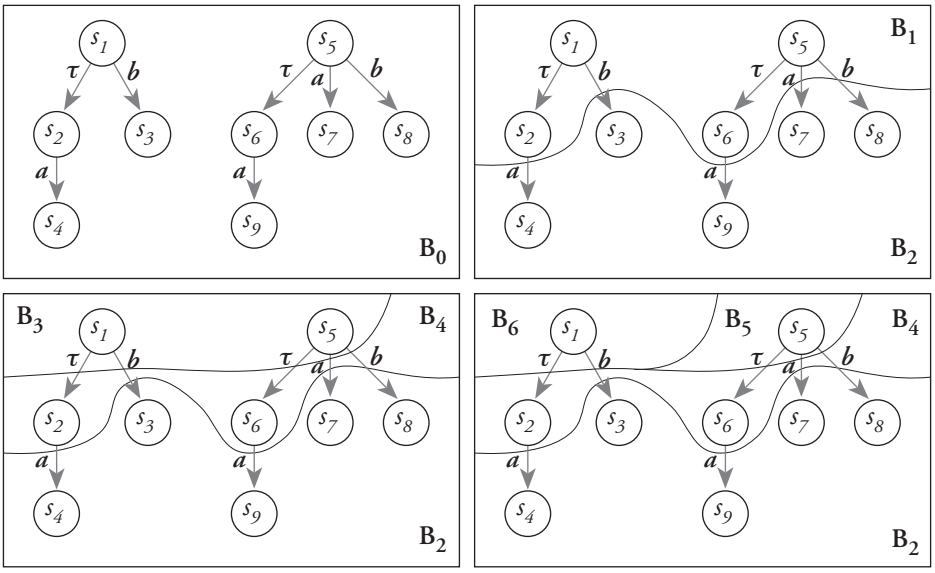


Figure 6.11 Block partitioning by *equiv*

6.6.3 CREDITS

The equivalence tester *equiv* in the PSF Toolkit is a C program written by E.E. Polak.

CHAPTER 7
CASE STUDY I: TEACHING ALGEBRAIC SPECIFICATIONS

WITH H.R. WALTERS

7.1 INTRODUCTION & MOTIVATION

This chapter is the description of an educational software package developed for, and used in a course on Software Engineering at the University of Amsterdam. Moreover it contains our experiences with its usage during the last three years.

In its search for ways to improve software quality and to put a halt to the so-called *software crisis*, Software Engineering has come up with many new techniques for software construction. Algebraic specification is one these techniques within the field of formal methods, that can be used to give a specification of the behaviour of a system. It aims at capturing the essential characteristics of a system at a high level of abstraction, rather than at obtaining an efficient program, and so it finds its use in the early stages of the software life cycle.

As part of the Software Engineering course at the University of Amsterdam, students have to learn the algebraic specification language ASF (Algebraic Specification Formalism) [BHK89]. To become better acquainted with the language, students have to get hands-on experience in writing specifications and executing them using the ASF implementation. The first two years this course was organized, students were given assignments to specify a medium-sized problem without any constraints on the format of the solution. The problems we encountered in following this approach were, that the interpretation and the elaboration of the assignments varied significantly from student to student, and that working with the ASF system, which is more or less a research tool, is not suitable for students. To overcome these problems we set out to develop a software package to support a practical assignment that had to meet several requirements.

The main requirement was that we wanted an assignment that left not much room for deviation in the elaboration. This also means that, because of the more uniform solutions,

marking them is easier. Eventually we came up with the idea to let the students specify a compiler. To increase attractiveness of the assignment, we decided that the compiler should use an imperative programming language as input, and wanted it to produce executable code.

We designed a high-level imperative programming language called Babble, which serves as the source language for the compiler, as well as the target language FOP, the machine code for a virtual stack machine. In this project we wanted to focus on the structural transformation between the languages as opposed to syntactical issues. In accordance, we have supplied a set of pre-processors and post-processors to perform the syntax analysis, type checking and machine code production, and so have created a more friendly interface to the ASF implementation. Furthermore, we supplied the students with the ASF signatures for both Babble and FOP and some standard data types to increase uniformity in the elaborations. This way we have created a so-called *software harness* for our students.

This chapter starts with the description of the algebraic specification language ASF, its differences with PSF, and the languages from the assignment: Babble and FOP. Then we discuss the different programs that make up the software environment, show a possible specification of the compiler and some of the problems students had to deal with. Finally, a detailed example of all the stages of the translation of a simple Babble program is given.

7.2 Deviations from standard ASF

Though ASF is part of the PSF language there are some slight differences between the implementation in the PSF Toolkit and the ASF language as described in [BHK89] and the text book used in the course [Bou91]. In this section we will focus on these differences.

As we have explained in earlier chapters, the PSF language supports two kinds of modules. Data types are specified in so-called *data modules*, which are derived from ASF, while processes are specified in *process modules*. Because ASF supports only one type of module, it is identified by the keyword *module*.

The next difference between both languages is that PSF has a larger number of reserved words than ASF. These reserved words cannot be used in an ASF specification that is to be processed with the PSF Toolkit. Furthermore, there are some constructs that are operators in PSF process specifications that are not allowed to be used as ASF operators. Below we give a list of the additional reserved words and operators in PSF.

atoms	hide	sets
communications	in	skip
data	merge	sum
definitions	of	to
encaps	process	|
for	processes	||

In standard ASF there is a built-in polymorphic conditional function *if*.

```
[X]  X = if(expr, a, b)          -- standard ASF
```

This function is not supported in the PSF implementation. A set of two conditional equations must be used instead.

```
[X1] X = a when expr = true      -- PSF implementation
[X2] X = b when expr = false
```

The final difference is that the PSF implementation does not allow for tupled output types.

```
functions                        -- standard ASF
  f : S1 # S2 -> S3 # S4
```

It is possible to overcome this omission by introducing extra sorts and constructors. For the example above, one has to introduce one new sort *S3-S4-tuple*, representing the tuple type *S3 # S4*, and one new constructor *make-S3-S4-tuple*.

```
sorts                            -- PSF implementation
  S3-S4-tuple
functions
  f : S1 # S2 -> S3-S4-tuple
  make-S3-S4-tuple : S3 # S4 -> S3-S4-tuple
```

Next all occurences of tuples *x # y* of type *S3 # S4* in the *equations* section should be replaced by the term *make-S3-S4-tuple(x,y)*.

7.3 BABBLE

In this section we will give the definition of Babble, the high-level language we use in this project. We start off with a global description and a motivation for the choice of language constructs. Next we will give the syntax definition and an example of a small program written in Babble.

7.3.1 BABBLE CONSTRUCTS

The main design issue for Babble has been to keep the language as simple as possible, yet to provide all the essential constructs to make it an imperative language in which it is possible to write small programs. Furthermore, we have tried to avoid duplication of instructions, because we wanted the diversity of the problems to be encountered in writing a compiler as large as possible. As a result there is only one *choice* statement and only one *loop* statement in Babble.

In Babble there is only one data type, namely the integers. Variables must be declared before they are used and are referred to by their names. There are no *procedures* or *functions* but there is a *block* construct in which local variables can be declared. If a *variable* name is declared, which already exists in a surrounding block, then the scope of the original variable is temporarily overridden. This is similar to the semantics of variables in inner blocks in Pascal [Wir71].

There are three simple statements in Babble. The simplest is the *skip* statement, which does nothing. Furthermore there is the *assignment* and a simple *output* statement. Babble provides two simple control structures: an *if-else-fi* and a *do-od* construct. The *else* clause in the *if-else-fi* is obligatory and should be filled with *skip* if no *else* clause is wanted. The

boolean expressions in the control structures use integers to represent booleans. The number *0* represents the boolean value *false*, and all other integers represent *true*. Below we will give the BNF grammar of Babble.

```
program
  : "program" IDENTIFIER block
  ;
block
  : '{' declarations ';' statement_list '}'
  ;
declarations
  : "var" id_list
  ;
id_list
  : IDENTIFIER
  | id_list ',' IDENTIFIER
  ;
statement_list
  : statement
  | statement_list ';' statement
  ;
statement
  : "skip"
  | IDENTIFIER ":=" expression
  | "if" condition statement_list "else" statement_list "fi"
  | "do" condition statement_list "od"
  | "print" expression
  | block
  ;
condition
  : '[' expression ']'
  ;
expression
  : simple_expr
  | simple_expr relop simple_expr
  ;
simple_expr
  : term
  | '-' term                                  -- minus sign
  | '~' term                                  -- logical negation
  | simple_expr addop term
  ;
term
  : factor
  | term mulop factor
  ;
factor
  : IDENTIFIER
  | NUMBER
  | "true"
  | "false"
  | '(' expression ')'
  ;
relop
  : '='                                       -- equality
  | "=/="                                     -- inequality
```

```
    | '<'                                          -- less than
    | '>'                                          -- greater than
    | "<="                                         -- less or equal
    | ">="                                         -- greater or equal
    ;
  addop
    : '+'                                          -- addition
    | '-'                                          -- subtraction
    | '|'                                          -- logical 'or'
    ;
  mulop
    : '*'                                          -- multiplication
    | '/'                                          -- integer division
    | '&'                                          -- logical 'and'
    | '%'                                          -- modulo
    ;
  IDENTIFIER
    : CHAR
    | IDENTIFIER CHAR
    ;
  NUMBER
    : DIGIT
    | NUMBER DIGIT
    ;
  CHAR
    : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
    | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
    | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
    ;
  DIGIT
    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    ;
  LAYOUT
    : ' ' | '\t' | '\n' | '\f' | '\r'              --tab, newline, form-feed,
    ;                                              --carriage return
```

7.3.2 AN EXAMPLE

In this section we will give an example of a simple Babble program that calculates the prime numbers between 1 and 1000. This example shows that it is possible to write a non-trivial program in Babble. In the course we have provided the students with this example, that embodies all the essential constructs from the Babble language, and its translation into FOP, generated by our compiler.

```
program primes
{
  var i,j,prime;

  i:= 2;
  do [i<=1000]
    j:= 2;
    prime := true;
    do [j*j <= i]
      if [(i%j) = 0]
        prime:= false
```

```
        else
          skip
        fi
        j:= j+1
      od
      if [ prime ]
        print i
      else
        skip
      fi;
      i:= i+1
    od
}
```

7.4 FOP

To be able to describe the compilation process we not only need a high-level source language, but also a low-level target language. For reasons of simplicity we also have defined our own low-level virtual machine code called FOP. The machine model on which FOP relies embodies three aspects. The main aspect is that FOP describes a machine with a stack to evaluate all expressions. Stack machine models allow for simple and small, yet elegant machine instruction sets, due to the absence of many different address modes.

The second part of our virtual machine is a simulated piece of random access memory to cater for the memory locations for the variables. Finally there is a list of numbered FOP instructions, representing the actual program. Next we will give a list of the instructions that are available in FOP.

HALT	halt the program
NOP	do nothing
PRINT	write the *top* of the stack to the output channel and remove *top*
MUL	multiply *top*-1 and *top* and push result on stack
DIV	divide *top*-1 by *top* (integer division) and push result on stack
MOD	calculate *top* modulo *top*-1 and push result on stack
ADD	add *top* to *top*-1 and push result on stack
SUB	subtract *top* from *top*-1 and push result on stack
PUSH n	push integer *n* on the stack
GET n	push the contents of memory location *n* onto the stack
STORE n	remove *top* from stack and put it into memory location *n*
JMP n	unconditional jump to instruction *n*
JMPN n	jump to instruction *n* if *top* is negative; remove *top* of stack
JMPZ n	jump to instruction *n* if *top* is equal to 0; remove *top* of stack

All instructions in FOP operate on the top of the stack or on the top of the stack and the element below. All operations are *destructive* with respect to the stack. This means that all operands needed for the evaluation of an instruction are popped from the stack, to be replaced with the result of the evaluation. FOP instructions consist of a keyword, or a keyword followed by a number. This number can refer to an integer, a memory location or an instruction in the numbered list of FOP instructions.

7.5 THE COMPUTER ENVIRONMENT

In this section we will describe the programs that constitute the Babble environment, the so-called *software harness* we have created. Furthermore we will describe the specifications of the standard data types and the signatures of Babble and FOP, that the students were given in advance.

7.5.1 THE COMPUTER PROGRAMS

A graphical representation of the Babble environment is given in Figure 7.1. The boxes represent programs and the ellipses represent specifications in the stated language. Files in the system are subject to a strict naming convention, which is clarified by the file names next to the ellipses.

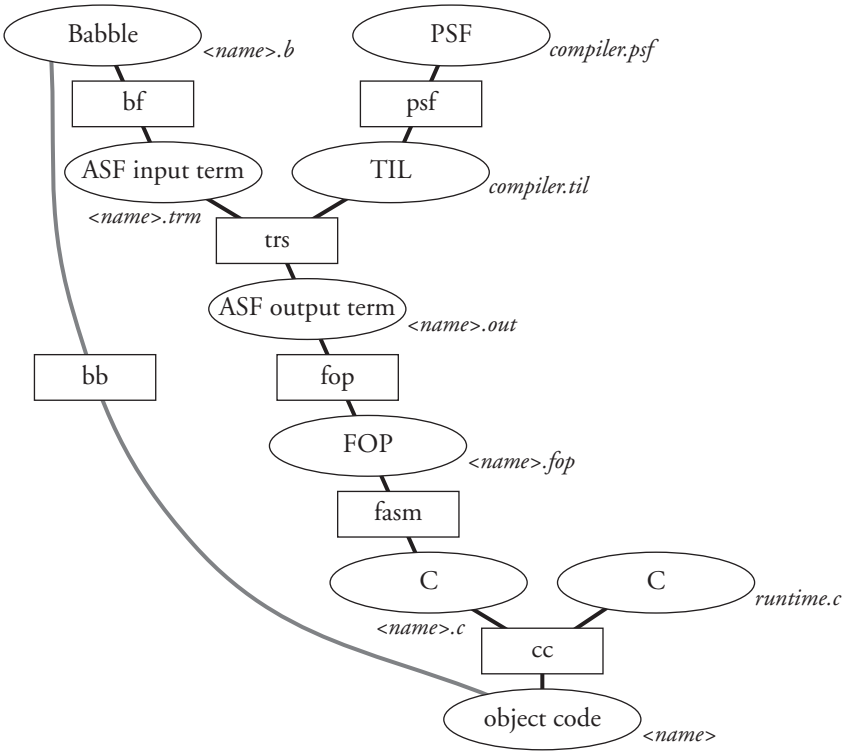


Figure 7.1 The Babble Computer Environment

The first program is called *bf* for 'Babble front-end'. It translates a Babble program (<name>.b) into an ASF input file (<name>.trm) to be processed by the term rewriter *trs*. The parser for *bf* was written using the Lex [LS79] and Yacc [Joh79] programs.

In the next phase the ASF specification written by the students transforms the algebraic version of Babble (<name>.trm) into the algebraic version of FOP (<name>.out). Execution of the algebraic specification is taken care for by the *trs* program.

The resulting output in the previous phase is one large ASF term in the FOP signature. This term is quite hard to read, so to make a more legible version of the FOP specification, the output of *trs* is transformed into FOP format (<name>.fop) by the *fop* program.

Finally the FOP code is translated into a C program (<name>.c), using *fasm*, and linked with a runtime system (runtime.c) that simulates the FOP processor. After compiling this completed program, using the C compiler, we end up with an executable version of the Babble program (<name>).

The Babble compiler script *bb* is a shortcut that executes the aforementioned programs in the right order.

7.5.2 THE STANDARD DATA TYPES

To let the students concentrate on the specification of the translation process only and to achieve more uniformity in the elaborations of the assignments, we supplied a library with some standard data types. These standard data types are: *booleans*, *binary numbers*, *decimal numbers* and *identifiers*. In the following sections we will list the specifications of these data types.

7.5.2.1 BOOLEANS

The definition of the *booleans* is straightforward.

```
data module Booleans
begin

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

  variables
    b : -> BOOLEAN

  equations
  [NOT1] not(true) = false
  [NOT2] not(false) = true
  [AND1] and(true, b) = b
  [AND2] and(false, b) = false
  [IOR1] or(true, b) = true
  [IOR2] or(false, b) = b

end Booleans
```

7.5.2.2 BINARY NUMBERS

Binary numbers are formed by sequences of binary digits (*bin-0, bin-1*). Constants in PSF cannot be overloaded, so the names *0* and *1* cannot be used for binary digits as well as decimal numbers. The two ^-operators are used to convert binary digits into binary numbers and to concatenate binary digits to binary numbers.

```
data module Bin-Numbers
begin

  exports
  begin
    sorts
     BIN-DIGIT, BIN-NUMBER
    functions
     bin-0 : -> BIN-DIGIT
     bin-1 : -> BIN-DIGIT
     ^_    : BIN-DIGIT -> BIN-NUMBER
     _^_   : BIN-NUMBER # BIN-DIGIT -> BIN-NUMBER
     eq    : BIN-DIGIT # BIN-DIGIT -> BOOLEAN
     eq    : BIN-NUMBER # BIN-NUMBER -> BOOLEAN
  end

  imports
    Booleans

  variables
    bd1, bd2 :-> BIN-DIGIT
    bn1, bn2 :-> BIN-NUMBER

  equations
  [BIN1] ^bin-0^bd1 = ^bd1
  [BIN2] eq(bin-0,bin-0) = true                -- equality binary digits
  [BIN3] eq(bin-0,bin-1) = false
  [BIN4] eq(bin-1,bin-1) = true
  [BIN5] eq(bin-1,bin-0) = false
  [BIN6] eq(^bd1,^bd2)       = eq(bd1,bd2)     -- equality binary numbers
  [BIN7] eq(^bd1,bn2^bd2)    = and(eq(bd1,bd2),eq(^bin-0,bn2))
  [BIN8] eq(bn1^bd1,^bd2)    = and(eq(bd1,bd2),eq(bn1,^bin-0))
  [BIN9] eq(bn1^bd1,bn2^bd2) = and(eq(bd1,bd2),eq(bn1,bn2))

end Bin-Numbers
```

7.5.2.3 DECIMAL NUMBERS

Decimal numbers are defined in a way similar to the binary numbers. For each digit a binary representation (*bin-rep*) is defined, which is used to define the equality on decimal digits in terms of the equality on binary numbers.

```
data module Numbers
begin

  exports
  begin
    sorts
     DIGIT, NUMBER
```

```
  functions
    0:-> DIGIT 1:-> DIGIT 2:-> DIGIT 3:-> DIGIT 4:-> DIGIT
    5:-> DIGIT 6:-> DIGIT 7:-> DIGIT 8:-> DIGIT 9:-> DIGIT
    ^_     : DIGIT -> NUMBER
    _^_    : NUMBER # DIGIT -> NUMBER
    eq     : DIGIT # DIGIT -> BOOLEAN
    eq     : NUMBER # NUMBER -> BOOLEAN
    bin-rep : DIGIT -> BIN-NUMBER
    s : NUMBER -> NUMBER
end

imports
  Booleans, Bin-Numbers

variables
  d1, d2 : -> DIGIT
  n1, n2 : -> NUMBER

equations
[NUM1] ^0^d1 = ^d1
[EQD1] eq(d1,d2) = eq(bin-rep(d1),bin-rep(d2))
[EQN1] eq(^d1,^d2)     = eq(d1,d2)
[EQN2] eq(^d1,n2^d2)   = and(eq(d1,d2),eq(^0,n2))
[EQN3] eq(n1^d1,^d2)   = and(eq(d1,d2),eq(n1,^0))
[EQN4] eq(n1^d1,n2^d2) = and(eq(d1,d2),eq(n1,n2))
[BRP0] bin-rep(0) = ^bin-0
[BRP1] bin-rep(1) = ^bin-1
[BRP2] bin-rep(2) = ^bin-1^bin-0
[BRP3] bin-rep(3) = ^bin-1^bin-1
[BRP4] bin-rep(4) = ^bin-1^bin-0^bin-0
[BRP5] bin-rep(5) = ^bin-1^bin-0^bin-1
[BRP6] bin-rep(6) = ^bin-1^bin-1^bin-0
[BRP7] bin-rep(7) = ^bin-1^bin-1^bin-1
[BRP8] bin-rep(8) = ^bin-1^bin-0^bin-0^bin-0
[BRP9] bin-rep(9) = ^bin-1^bin-0^bin-0^bin-1
[SCS0] s(^0) = ^1
[SCS1] s(^1) = ^2
[SCS2] s(^2) = ^3
[SCS3] s(^3) = ^4
[SCS4] s(^4) = ^5
[SCS5] s(^5) = ^6
[SCS6] s(^6) = ^7
[SCS7] s(^7) = ^8
[SCS8] s(^8) = ^9
[SCS9] s(^9) = ^1^0
[SCC0] s(n1^0) = n1^1
[SCC1] s(n1^1) = n1^2
[SCC2] s(n1^2) = n1^3
[SCC3] s(n1^3) = n1^4
[SCC4] s(n1^4) = n1^5
[SCC5] s(n1^5) = n1^6
[SCC6] s(n1^6) = n1^7
[SCC7] s(n1^7) = n1^8
[SCC8] s(n1^8) = n1^9
[SCC9] s(n1^9) = s(n1)^0

end Numbers
```

7.5.2.4 IDENTIFIERS

Identifiers in a Babble program are represented as strings of characters. The definition of the sort *ID* is similar to the definition of the binary numbers and decimal numbers.

```
data module Ids
begin

  exports
  begin
    sorts
     CHAR, ID
    functions
     a:-> CHAR b:-> CHAR c:-> CHAR d:-> CHAR e:-> CHAR
     f:-> CHAR g:-> CHAR h:-> CHAR i:-> CHAR j:-> CHAR
     k:-> CHAR l:-> CHAR m:-> CHAR n:-> CHAR o:-> CHAR
     p:-> CHAR q:-> CHAR r:-> CHAR s:-> CHAR t:-> CHAR
     u:-> CHAR v:-> CHAR w:-> CHAR x:-> CHAR y:-> CHAR
     z:-> CHAR
     ^_    : CHAR -> ID
     _^_   : ID # CHAR -> ID
     eq    : CHAR # CHAR -> BOOLEAN
     eq    : ID # ID -> BOOLEAN
     bin-rep : CHAR -> BIN-NUMBER
  end

  imports
   Bin-Numbers

  variables
   c1, c2 : -> CHAR
   s1, s2 : -> ID

  equations
  [IDS1] eq(c1,c2) = eq(bin-rep(c1),bin-rep(c2))    -- character equality
  [IDS2] eq(^c1,^c2)     = eq(c1,c2)               -- identifier equality
  [IDS3] eq(^c1,s2^c2)   = false
  [IDS4] eq(s1^c1,^c2)   = false
  [IDS5] eq(s1^c1,s2^c2) = and(eq(c1,c2),eq(s1,s2))
  [BRPA] bin-rep(a) = ^bin-0                   -- character representation
  [BRPB] bin-rep(b) = ^bin-1
  [BRPC] bin-rep(c) = ^bin-1^bin-0
  [BRPD] bin-rep(d) = ^bin-1^bin-1
  [BRPE] bin-rep(e) = ^bin-1^bin-0^bin-0
  [BRPF] bin-rep(f) = ^bin-1^bin-0^bin-1
  [BRPG] bin-rep(g) = ^bin-1^bin-1^bin-0
  [BRPH] bin-rep(h) = ^bin-1^bin-1^bin-1
  [BRPI] bin-rep(i) = ^bin-1^bin-0^bin-0^bin-0
  [BRPJ] bin-rep(j) = ^bin-1^bin-0^bin-0^bin-1
  [BRPK] bin-rep(k) = ^bin-1^bin-0^bin-1^bin-0
  [BRPL] bin-rep(l) = ^bin-1^bin-0^bin-1^bin-1
  [BRPM] bin-rep(m) = ^bin-1^bin-1^bin-0^bin-0
  [BRPN] bin-rep(n) = ^bin-1^bin-1^bin-0^bin-1
  [BRPO] bin-rep(o) = ^bin-1^bin-1^bin-1^bin-0
  [BRPP] bin-rep(p) = ^bin-1^bin-1^bin-1^bin-1
  [BRPQ] bin-rep(q) = ^bin-1^bin-0^bin-0^bin-0^bin-0
  [BRPR] bin-rep(r) = ^bin-1^bin-0^bin-0^bin-0^bin-1
```

```
  [BRPS] bin-rep(s) = ^bin-1^bin-0^bin-0^bin-1^bin-0
  [BRPT] bin-rep(t) = ^bin-1^bin-0^bin-0^bin-1^bin-1
  [BRPU] bin-rep(u) = ^bin-1^bin-0^bin-1^bin-0^bin-0
  [BRPV] bin-rep(v) = ^bin-1^bin-0^bin-1^bin-0^bin-1
  [BRPW] bin-rep(w) = ^bin-1^bin-0^bin-1^bin-1^bin-0
  [BRPX] bin-rep(x) = ^bin-1^bin-0^bin-1^bin-1^bin-1
  [BRPY] bin-rep(y) = ^bin-1^bin-1^bin-0^bin-0^bin-0
  [BRPZ] bin-rep(z) = ^bin-1^bin-1^bin-0^bin-0^bin-1

end Ids
```

7.5.3 LANGUAGE SIGNATURES

The signatures of the internal ASF representation of the terms of the two languages Babble and FOP are given in the following sections.

7.5.3.1 BABBLE SIGNATURE

The Babble signature is relatively simple. The two most interesting sorts are the expressions (*EXP*) and statements (*STAT*). A list of statements (*STAT-LIST*) is coupled with a list of identifiers (*ID-LIST*) representing variables to form a *BLOCK*. Remember that Babble is a block-oriented language, so a block can in turn be interpreted as a statement. Finally a *PROGRAM* is constructed by combining a block and an identifier (*ID*) representing the program name.

```
data module Babble
begin

  exports
  begin
    sorts
     PROGRAM, BLOCK, ID-LIST, EXP-LIST, STAT-LIST, STAT, EXP
    functions
     program : ID # BLOCK -> PROGRAM
     block : ID-LIST # STAT-LIST -> BLOCK
     mt-ids : -> ID-LIST
     ids : ID-LIST # ID -> ID-LIST
     mt-exps : -> EXP-LIST
     exps : EXP-LIST # EXP -> EXP-LIST
     mt-stats : -> STAT-LIST
     stats : STAT-LIST # STAT -> STAT-LIST

     skp : -> STAT
     assign : ID # EXP -> STAT
     par-assign : ID-LIST # EXP-LIST -> STAT
     if-fi : EXP # STAT-LIST # STAT-LIST -> STAT
     do-od : EXP # STAT-LIST -> STAT
     print : EXP -> STAT
     block : BLOCK -> STAT

     eq : EXP # EXP -> EXP
     lt : EXP # EXP -> EXP
     gt : EXP # EXP -> EXP
     neq : EXP # EXP -> EXP
     leq : EXP # EXP -> EXP
     geq : EXP # EXP -> EXP
```

```
        add : EXP # EXP -> EXP
        sub : EXP # EXP -> EXP
        mul : EXP # EXP -> EXP
        div : EXP # EXP -> EXP
        mod : EXP # EXP -> EXP

        or : EXP # EXP -> EXP
        and : EXP # EXP -> EXP

        minus : EXP -> EXP
        neg : EXP -> EXP

        factor : ID -> EXP
        factor : NUMBER -> EXP

        b-true : -> EXP
        b-false : -> EXP
    end

    imports
      Booleans, Ids, Numbers

end Babble
```

7.5.3.2 FOP SIGNATURE

The signature for the FOP terms is given below. The individual FOP instructions are represented by the sort *FOPCODE*. A FOP program consists of a list of instructions and is represented by the sort *FOP*.

```
data module Fop
begin

  exports
  begin
    sorts
      FOPCODE, FOP
    functions
      halt:          -> FOPCODE
      nop:           -> FOPCODE
      push:  NUMBER -> FOPCODE
      print:         -> FOPCODE
      mul:           -> FOPCODE
      div:           -> FOPCODE
      mod:           -> FOPCODE
      add:           -> FOPCODE
      sub:           -> FOPCODE
      get:   NUMBER -> FOPCODE
      store: NUMBER -> FOPCODE
      jmp:   NUMBER -> FOPCODE
      jmpz:  NUMBER -> FOPCODE
      jmpn:  NUMBER -> FOPCODE

      mt-fop: -> FOP
      fop: FOPCODE # FOP -> FOP
  end
```

```
    imports
      Numbers

end Fop
```

7.6 A NAIVE BABBLE COMPILER

In this project, the emphasis was on teaching algebraic techniques for software construction and functional correctness rather than on efficiency of the compiler or of the generated code. Note that students did not have any background in compiler construction. For this reason, but also to keep the assignment reasonably small, syntax-checking and type-checking were not considered. That is, compilers only needed to compile statically correct programs.

In this section we will present an implementation of a naive Babble compiler, and discuss some of its features. It is not our aim to discuss the complete functionality of the compiler here, but more to give an idea of the complexity of the problems that the students had to solve.

In a Babble compiler, only two problems are less than straightforward: memory allocation for variables and address calculation for forward jumps. Since Babble does not have functions or procedures, the simplest form of memory allocation can be used. This means that every variable (unique name and scope) is assigned a unique address. Some students implemented optimization here: variables with non-overlapping scopes can share memory locations. In our compiler we will not consider this optimization.

The second problem, *forward jumps*, can be approached in two ways. In the first approach the compiler makes two passes over the code. In the first pass temporary labels are generated for the destinations of forward jumps which must then be resolved in a second pass of the compiler. In the second approach, compilation is performed bottom-up, and jumps can be computed immediately, by considering the length of the generated code. Both approaches were used by students. In this chapter we will define a two-pass compiler. This compiler consists of five main modules: *Babble-Compiler*, *Pass1*, *Pass2*, *Symtabs* and *Adrtabs*. We will explain these modules in more detail in the following sections.

7.6.1 THE ADDRESS TABLE

Module *Adrtabs* implements the functions for the address table. The address table is used in the calculation of the actual addresses of the labels generated in forward jumps.

Entries to the address table are added using the function *add()*. The first argument is an identification number for a *label*, the second argument the *actual address* and the third argument the *address table* to which the new entry gets attached. The actual address can be retrieved later by means of the function *lookup()*. If *lookup()* cannot find a label it returns address *0*.

```
data module Adrtabs
begin
  exports
  begin
    sorts
      ADRTAB
```

```
    functions
     mt-adrtab: -> ADRTAB
     add: NUMBER # NUMBER # ADRTAB -> ADRTAB
     lookup: NUMBER # ADRTAB -> NUMBER
  end
  imports
    Numbers
  variables
    label,label',address: -> NUMBER
    address-table: -> ADRTAB

  equations
  [LKP1] lookup(label,mt-adrtab) = ^0
  [LKP2] lookup(label,add(label',address,address-table)) = address  when
           eq(label,label') = true
  [LKP3] lookup(label,add(label',address,address-table)) =
          lookup(label,address-table)  when
           eq(label,label') = false

end Adrtabs
```

7.6.2 SYMBOL TABLES

In module *Symtabs,* symbol tables are defined. A symbol table is a stack of frames, coinciding with the nesting structure of blocks. Each frame is a list of variable-location pairs. The module exports a function *lookup(),* which gets a symbol table and a variable name, and returns the location for that variable name in the closest surrounding block.

```
data module Symtabs
begin

  exports
  begin
    sorts
     SYMTAB
    functions
     mt-symtab:                      -> SYMTAB
     add-frame: SYMTAB               -> SYMTAB
     del-frame: SYMTAB               -> SYMTAB
     insert:    ID # NUMBER # SYMTAB -> SYMTAB
     lookup:    ID # SYMTAB          -> NUMBER
     has-local: ID # SYMTAB          -> BOOLEAN
     has:       ID # SYMTAB          -> BOOLEAN
  end

  imports
    Numbers, Ids

  sorts
    FRAME
```

```
  functions
    mt-frame:                       -> FRAME
    add:       ID # NUMBER # FRAME -> FRAME
    has:       ID # FRAME           -> BOOLEAN
    add:       FRAME # SYMTAB       -> SYMTAB
    lookup:    ID # FRAME           -> NUMBER

  variables
    symtab: -> SYMTAB
    id, id': -> ID
    num: -> NUMBER
    fr: -> FRAME

  equations
  [ADDF] add-frame(symtab) = add(mt-frame,symtab)
  [DLF1] del-frame(add(fr,symtab)) = symtab
  [DLF2] del-frame(mt-symtab) = mt-symtab
  [INS1] insert(id,num,mt-symtab) = mt-symtab
  [INS2] insert(id,num,add(fr,symtab)) = add(add(id,num,fr),symtab)
  [LKP1] lookup(id,mt-symtab) = ^0
  [LKP2] lookup(id,add(fr,symtab)) = lookup(id,fr)  when
           has(id,fr) = true
  [LKP3] lookup(id,add(fr,symtab)) = lookup(id,symtab)  when
           has(id,fr) = false
  [HLC1] has-local(id,mt-symtab) = false
  [HLC2] has-local(id,add(fr,symtab)) = has(id,fr)
  [HAS1] has(id,mt-symtab) = false
  [HAS2] has(id,add(fr,symtab)) = true  when
           has(id,fr) = true
  [HAS3] has(id,add(fr,symtab)) = has(id,symtab)  when
           has(id,fr) = false
  [HAS4] has(id,mt-frame) = false
  [HAS5] has(id,add(id',num,fr)) = true  when
           eq(id,id') = true
  [HAS6] has(id,add(id',num,fr)) = has(id,fr)  when
           eq(id,id') = false
  [LKP4] lookup(id,mt-frame) = ^0
  [LKP5] lookup(id,add(id',num,fr)) = num  when
           eq(id,id') = true
  [LKP6] lookup(id,add(id',num,fr)) = lookup(id,fr)  when
           eq(id,id') = false

end Symtabs
```

7.6.3 THE ADDITIONAL FORWARD JUMPS

Two extra forward jump FOP codes, representing the temporary labels generated during the first pass of the compiler, are defined in module *Label-Fop*. Furthermore, the function *fop()* is defined to attach the extra FOP codes to a list of FOP codes.

```
data module Label-Fop
begin

  exports
  begin
    sorts
      LABEL-FOP
```

```
    functions
      jmpfwd: NUMBER -> LABEL-FOP
      jmpzfwd: NUMBER -> LABEL-FOP
      fop : LABEL-FOP # FOP -> FOP
  end

  imports
    Fop

end Label-Fop
```

7.6.4 THE FIRST PASS

Module *Pass1* is the heart of the compiler. It exports a number of compiling functions, all called *pass1*. These functions get, as arguments, a part of a Babble program, the FOP code compiled so far, the address of the next FOP instruction to be generated, the current symbol table, the first free memory location (for variables), the address table, as explained, and the first free label. These functions return the created FOP code, and all updated counters and tables, with the exception of the symbol table, which can only grow within a block, but not beyond it.

In the first pass, forward jumps are compiled using the special *forward jump* FOP-code. Furthermore, an address table is created in which these labels are associated with the corresponding address.

We will intersperse the following listing of module *Pass1* with comments explaining some of the equations.

```
data module Pass1
begin

  exports
  begin
    sorts
      TUP
    functions
      -- p1-tup is used to create tupled output
      p1-tup: FOP # NUMBER # NUMBER # ADRTAB # NUMBER -> TUP

      -- the functions implementing the first compilation pass
      pass1: BLOCK # FOP # NUMBER # SYMTAB # NUMBER # ADRTAB # NUMBER
          -> TUP
      pass1: STAT-LIST # FOP # NUMBER # SYMTAB # NUMBER # ADRTAB # NUMBER
          -> TUP
      pass1: STAT # FOP # NUMBER # SYMTAB # NUMBER # ADRTAB # NUMBER
          -> TUP
      pass1: EXP # FOP # NUMBER # SYMTAB # NUMBER # ADRTAB # NUMBER
          -> TUP
  end

  imports
    Booleans, Babble, Symtabs, Fop, Label-Fop, Adrtabs

  sorts
    SYMTAB-NUMBER
```

```
functions
  binop: EXP # EXP # FOPCODE -> EXP
  addvars: ID-LIST # SYMTAB # NUMBER -> SYMTAB-NUMBER
  av-tup: SYMTAB # NUMBER -> SYMTAB-NUMBER

variables
  id: -> ID
  ids: -> ID-LIST
  block: -> BLOCK
  fop, fop1, fop2, fop3, fop4: -> FOP
  adrtab, adrtab1, adrtab2, adrtab3, adrtab4: -> ADRTAB
  adr, adr1, adr2, adr3, adr4: -> NUMBER
  vcnt, vcnt1, vcnt2, vcnt3, vcnt4: -> NUMBER
  acnt, acnt1, acnt2, acnt3, acnt4: -> NUMBER
  vars: -> ID-LIST
  stats, tstats, estats: -> STAT-LIST
  symtab, symtab1, symtab2: -> SYMTAB
  stat: -> STAT
  num: -> NUMBER
  exp, exp1, exp2: -> EXP
  exps: -> EXP-LIST
  op: -> FOPCODE

equations
```

As a first example, consider the simplest equation:

```
[SKIP] pass1(skp,fop,adr,symtab,vcnt,adrtab,acnt) =
       p1-tup(fop(nop,fop), s(adr), vcnt, adrtab, acnt)
```

In this example *skp* is a Babble statement, *nop* is a FOP code, and all other plain identifiers are variables. The function *fop()* strings together FOP codes.

Observe that the last code is added on top. This means that eventually the FOP program is obtained in reverse order. This is not a problem, because the order is reversed again in the second pass of the compiler.

Also observe that all counters remain the same, except the current address, which is increased by one.

```
[ASGN] pass1(assign(id,exp),fop1,adr1,symtab,vcnt1,adrtab1, acnt1) =
       p1-tup(fop(store(num),fop2),s(adr2),vcnt2,adrtab2,acnt2)  when
        pass1(exp,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
         p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2),
        lookup(id,symtab) = num
[PAS1] pass1(par-assign(mt-ids,mt-exps),fop,adr,
          symtab,vcnt,adrtab,acnt)=
       p1-tup(fop, adr, vcnt, adrtab, acnt)
[PAS2] pass1(par-assign(ids(ids,id),exps(exps,exp)),
          fop1,adr1,symtab,vcnt1,adrtab1, acnt1) =
       p1-tup(fop(store(num),fop3),s(adr3),vcnt3,adrtab3,acnt3)  when
        pass1(exp,fop1,adr1,symtab,vcnt1,adrtab1, acnt1) =
         p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2),
        pass1(par-assign(ids,exps),fop2,adr2,
            symtab,vcnt2,adrtab2,acnt2) =
         p1-tup(fop3, adr3, vcnt3, adrtab3, acnt3),
        lookup(id,symtab) = num
```

```
[PRNT] pass1(print(exp),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        p1-tup(fop(print,fop2), s(adr2), vcnt2, adrtab2, acnt2)  when
         pass1(exp,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
          p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2)
[SBLK] pass1(block(block),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(block,fop,adr,symtab,vcnt,adrtab,acnt)
```

As a second example we will look at the compilation of operators. There are three classes of operators: arithmetic, boolean, and relational. All arithmetic operators are handled the same. For instance, the following equation compiles additions.

```
[ADDX] pass1(add(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,add),fop,adr,symtab,vcnt,adrtab,acnt)
```

The constant *add,* used in the right-hand side is the FOP code to use for this binary operator.

```
[SUBX] pass1(sub(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,sub),fop,adr,symtab,vcnt,adrtab,acnt)
[MULX] pass1(mul(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,mul),fop,adr,symtab,vcnt,adrtab,acnt)
[DIVX] pass1(div(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,div),fop,adr,symtab,vcnt,adrtab,acnt)
[MODX] pass1(mod(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,mod),fop,adr,symtab,vcnt,adrtab,acnt)
[OREX] pass1(or(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,add),fop,adr,symtab,vcnt,adrtab,acnt)
[ANDX] pass1(and(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(binop(exp1,exp2,mul),fop,adr,symtab,vcnt,adrtab,acnt)
```

The notion *binop(),* used in the compilation of arithmetic operators, is compiled by the following (conditional) equation.

```
[BNOP] pass1(binop(exp1,exp2,op),fop1,adr1,
           symtab,vcnt1,adrtab1,acnt1) =
        p1-tup(fop(op,fop3), s(adr3), vcnt3, adrtab3, acnt3)  when
         pass1(exp1,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
          p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2),
         pass1(exp2,fop2,adr2,symtab,vcnt2,adrtab2,acnt2) =
          p1-tup(fop3, adr3, vcnt3, adrtab3, acnt3)
```

In this fashion the arithmetic operators, for which a machine equivalent exists, are handled.
In the following examples we will look at the relational operators. Below, the equation specifying the translation of the *less than*-operator (*lt*) is shown.

```
[LTEX] pass1(lt(exp1,exp2),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        p1-tup(fop(push(^1),
            fop(jmp(s(s(s(s(adr2))))),
            fop(push(^0),
            fop(jmpn(s(s(s(adr2)))),fop2)))),
            s(s(s(s(adr2)))), vcnt2, adrtab2, acnt2)  when
         pass1(sub(exp1,exp2),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
          p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2)
```

In order to understand this equation, consider the following table. First, code is generated for the subtraction of the two expressions. Then code is added which, depending on the sign of the result, pushes *0* or *1* on the stack.

```
        …                   { Code for subtraction of exp1 by exp2 }
$:      JMPN    $+3
        PUSH    0
        JMP     $+4
        PUSH    1
```

Now, other relational operators are handled using this *lt* operator.

```
[GTEX] pass1(gt(exp1,exp2),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        pass1(lt(exp2,exp1),fop1,adr1,symtab,vcnt1,adrtab1,acnt1)
[GEQX] pass1(geq(exp1,exp2),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        pass1(neg(lt(exp1,exp2)),fop1,adr1,symtab,vcnt1,adrtab1,acnt1)
[LEQX] pass1(leq(exp1,exp2),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        pass1(neg(lt(exp2,exp1)),fop1,adr1,symtab,vcnt1,adrtab1,acnt1)
[MINX] pass1(minus(exp),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        pass1(sub(factor(^0),exp),fop1,adr1,symtab,vcnt1,adrtab1,acnt1)

[EQUX] pass1(eq(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(neg(sub(exp1,exp2)),fop,adr,symtab,vcnt,adrtab,acnt)
[NEQX] pass1(neq(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt) =
        pass1(sub(exp1,exp2),fop,adr,symtab,vcnt,adrtab,acnt)
[NEGX] pass1(neg(exp),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        p1-tup(fop(push(^1),
            fop(jmp(s(s(s(s(adr2))))),
            fop(push(^0),
            fop(jmpz(s(s(s(adr2)))),fop2)))),
            s(s(s(s(adr2)))), vcnt2, adrtab2, acnt2)  when
         pass1(exp,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
          p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2)
[FAC1] pass1(factor(id),fop,adr,symtab,vcnt,adrtab,acnt) =
        p1-tup(fop(get(lookup(id,symtab)),fop),s(adr),vcnt,adrtab,acnt)
[FAC2] pass1(factor(num),fop,adr,symtab,vcnt,adrtab,acnt) =
        p1-tup(fop(push(num),fop), s(adr), vcnt, adrtab, acnt)
[BLN1] pass1(b-true,fop,adr,symtab,vcnt,adrtab,acnt) =
        p1-tup(fop(push(^1),fop), s(adr), vcnt, adrtab, acnt)
[BLN2] pass1(b-false,fop,adr,symtab,vcnt,adrtab,acnt) =
        p1-tup(fop(push(^0),fop), s(adr), vcnt, adrtab, acnt)
```

As an example of control structures we will look at the *do*-statement.

```
[DOOD] pass1(do-od(exp,stats),fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
        p1-tup(fop(jmp(adr1),fop3), s(adr3), vcnt3,
            add(acnt2,s(adr3),adrtab3), acnt3)  when
         pass1(exp,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
          p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2),
         pass1(stats,fop(jmpzfwd(acnt2),fop2),s(adr2),
             symtab,vcnt2,adrtab2,s(acnt2)) =
          p1-tup(fop3, adr3, vcnt3, adrtab3, acnt3)
```

In the first condition, code is generated for the expression that guards the loop. Then a forward jump is compiled with label *acnt2,* and code is generated for the statements in the

second condition. Finally, a backward jump is added in the right-hand side of the main equation. The address for this jump is already known, so an ordinary jump is generated.

```
adr1:   …                                { Code for evaluating the loop condition: exp }
        JMPZFWD  <acnt2,s(adr3)>
        …                                { Code for the statements: stats }
adr3:   JMP      adr1
```

Observe that after evaluation of the two conditions the target address of the forward jump becomes available. Unfortunately the forward jump cannot be altered at this point. In the algebraic framework, it is not possible to compile a normal jump with an unbound variable, which can then be bound afterwards (as it would be, for instance, in Prolog [CM81]). In our situation we simply add the label and the actual address to the address table (*add(acnt2,s(adr3),adrtab3)*), to be used in the second pass of the compiler. The compilation of the *if*-statement is performed in a similar fashion.

```
[IFFI] pass1(if-fi(exp,tstats,estats),fop1,adr1,symtab,
           vcnt1,adrtab1,acnt1) =
        p1-tup(fop4,adr4,vcnt4,add(acnt3,adr4,adrtab4),s(acnt4))  when
         pass1(exp,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
          p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2),
         pass1(tstats,fop(jmpzfwd(acnt2),fop2),s(adr2),
             symtab,vcnt2,adrtab2,s(acnt2)) =
          p1-tup(fop3, adr3, vcnt3, adrtab3, acnt3),
         pass1(estats,fop(jmpfwd(acnt3),fop3),s(adr3),
             symtab,vcnt3,add(acnt2,s(adr3),adrtab3),s(acnt3)) =
          p1-tup(fop4, adr4, vcnt4, adrtab4, acnt4)

[ADV1] addvars(mt-ids,symtab,vcnt) = av-tup(symtab,vcnt)
[ADV2] addvars(ids(ids,id),symtab1,vcnt) =
        addvars(ids,symtab2,s(vcnt))  when
         insert(id,vcnt,symtab1) = symtab2
[STL1] pass1(mt-stats,fop,adr,symtab,vcnt1,adrtab,acnt) =
        p1-tup(fop, adr, vcnt1, adrtab, acnt)
[STL2] pass1(stats(stats,stat),fop1,adr1,
           symtab,vcnt1,adrtab1,acnt1) =
        pass1(stat,fop2,adr2,symtab,vcnt2,adrtab2,acnt2)  when
          pass1(stats,fop1,adr1,symtab,vcnt1,adrtab1,acnt1) =
           p1-tup(fop2, adr2, vcnt2, adrtab2, acnt2)
```

Finally we will present the equation concerning *program blocks,* which is at this point very straightforward.

```
[VBLK] pass1(block(vars,stats),fop,adr,symtab1,vcnt1,adrtab,acnt) =
        pass1(stats,fop,adr,symtab2,vcnt2,adrtab,acnt)  when
         addvars(vars,add-frame(symtab1),vcnt1) = av-tup(symtab2,vcnt2)

end Pass1
```

7.6.5 THE SECOND PASS

In module *Pass2* the list of FOP instructions is reversed again and all symbolic forward jumps are substituted with the correct actual jump. The addresses for the labels used in forward jumps are stored in the *address table* and are retrieved by the function *lookup()*.

```
data module Pass2
begin

  exports
  begin
    functions
     pass2:  FOP # ADRTAB    -> FOP
  end

  imports
    Booleans, Numbers, Fop, Label-Fop, Adrtabs
  functions
    compute:  FOP # FOP # ADRTAB      -> FOP
  variables
    fop, fop1, fop2: -> FOP
    adrtab: -> ADRTAB
    num: -> NUMBER
    fopc: -> FOPCODE
  equations
  [2NDP] pass2(fop,adrtab) = compute(fop, fop(halt,mt-fop), adrtab)
  [CMP1] compute(mt-fop,fop,adrtab) = fop
  [CMP2] compute(fop(jmpfwd(num),fop1),fop2,adrtab) =
          compute(fop1,fop(jmp(lookup(num,adrtab)),fop2),adrtab)
  [CMP3] compute(fop(jmpzfwd(num),fop1),fop2,adrtab) =
          compute(fop1,fop(jmpz(lookup(num,adrtab)),fop2),adrtab)
  [CMP4] compute(fop(fopc,fop1),fop2,adrtab) =
          compute(fop1,fop(fopc,fop2),adrtab)

end Pass2
```

7.6.6 THE COMPILER

Module *Babble-Compiler* imports *Pass1* and *Pass2* (and indirectly all other modules), and defines the *compile* function by calling the two passes with the correct initialization.

```
data module Babble-Compiler
begin
  exports
  begin
    functions
     compile: PROGRAM -> FOP
  end
  imports
    Pass1, Pass2
  variables
    id: -> ID
    block: -> BLOCK
    fop: -> FOP
    adr,vcnt,acnt: -> NUMBER
    adrtab: -> ADRTAB
```

```
  equations
  [CMP1] compile(program(id,block)) = pass2(fop,adrtab)  when
           pass1(block,mt-fop,^0,mt-symtab,^1,mt-adrtab,^0) =
            p1-tup(fop, adr, vcnt, adrtab, acnt)

end Babble-Compiler
```

7.6.7 SOME REMARKS

We have defined a simple Babble compiler, which is functionally correct. No real efficiency can be expected from this compiler, partly because of the straightforward way in which it was written, and partly because of the inherent inferior execution time of algebraic implementations. Nevertheless our approach displays several interesting qualities.

- The compiler is very compact: less than 60 equations, less than 250 lines of text. An experienced (algebraic) programmer can design and implement this compiler in two days. We think this to be a reasonable size for an assignment in a practical course, although we must mention that most students came up with larger specifications.
- The compiler is reasonably easy to maintain and extend. For example, students were asked to add a new feature to Babble, a *parallel assignment* statement, which did not occur in the original language definition. This involves addition of one or two auxiliary functions, and about six equations (included in the 60 mentioned above), and can be done in a couple of hours.
- The modular structure, the high level of abstraction, and the uniform interface make sure that the students are able to focus on the key issues of solving the problems at hand, without the danger of being distracted by implementation details, which would be the case were this project carried out using a traditional imperative language.

7.7 AN EXAMPLE

To give an impression of what formats are encountered during the complete trajectory from Babble to FOP, we will study the stages in the translation of one program in detail. This simple example program will print the squares of the numbers from 1 up to 10. We start with the Babble program:

```
program example
{
  var i;

  i:= 1;
  do [ i <= 10 ]
    print i*i;
    i:= i+1
  od
}
```

This is transformed into input for the term rewriter, using the *bf* program.

```
compile(program(^e^x^a^m^p^l^e,block(ids(mt-ids,^i),stats(stats(mt-stats,
assign(^i,factor(^1))),do-od(leq(factor(^i),factor(^1^0)),stats(stats( mt-
stats,print(mul(factor(^i),factor(^i)))),assign(^i,add(factor(^i),
factor(^1)))))))))
```

Now that we have a version of the program in ASF using the Babble signature, we can transform it, by means of the algebraic specification of the compiler, into an ASF version using the FOP signature :

```
compile(program(…))   = fop(push((^ 1)),fop(store((^ 1)),fop(push(((^ 1) ^
0)),fop(get((^ 1)),fop(sub,fop(jmpn((^ 8)),fop(push((^ 0)),fop(jmp((
^ 9)),fop(push((^ 1)),fop(jmpz(((^ 1) ^ 2)),fop(push((^ 0)),fop(jmp(((
^ 1) ^ 3)),fop(push((^ 1)),fop(jmpz(((^ 2) ^ 3)),fop(get((^ 1)),fop(get((
^ 1)),fop(mul,fop(print,fop(get((^ 1)),fop(push((^ 1)),fop(add,fop(store(
(^ 1)),fop(jmp((^ 2)),fop(halt,mt-fop))))))))))))))))))))))
```

Finally we can transform the ASF output into FOP using the *fop* program for better readability:

```
     BEGIN
0:   PUSH   1
1:   STORE  1
2:   PUSH   10
3:   GET    1
4:   SUB
5:   JMPN   8
6:   PUSH   0
7:   JMP    9
8:   PUSH   1
9:   JMPZ   12
10:  PUSH   0
11:  JMP    13
12:  PUSH   1
13:  JMPZ   23
14:  GET    1
15:  GET    1
16:  MUL
17:  PRINT
18:  GET    1
19:  PUSH   1
20:  ADD
21:  STORE  1
22:  JMP    2
23:  HALT
     END
```

7.8 CONCLUSIONS

We have presented a software package developed to teach algebraic specifications in the context of a Software Engineering course and have shown how the tools from the PSF Toolkit have been integrated into this software environment. Our main aim has been to create an interesting non-trivial assignment and an accompanying *software harness* to be able to focus on the essentials of algebraic specifications.

We have designed an assignment in which students have to specify the algebraic *heart* of a compiler. To support this assignment we have designed a high-level language called Babble, and a low-level language called FOP. Furthermore, we have implemented a set of programs that interface Babble and FOP to the PSF Toolkit:

- a syntax and type checker for Babble programs
- a translator that translates Babble programs into terms that can be interpreted by the the term rewriter from the PSF Toolkit.
- a code generator that generates a C program from the output of the term rewriter
- a run time system that is to be linked with the generated C program to form an executable program.

We have experienced that being able to actually run the specified compiler, and being able to generate executable code, increased the attractiveness of the assignment for the students. Moreover, we have succeeded in generating an environment in which comparing and marking the elaborations has become easier.

The two major disadvantages of algebraic implementations are the poor execution speed, and the absence of a versatile user interface. The first problem is of less importance in the context of educational systems and prototyping, and should be weighed against the gain in efficiency of the overall software development. The second problem has been discussed in this article. We have shown that conventional tools such as Lex [LS79] and Yacc [Joh79] can be used to create a software harness interfacing to the term rewriter and the PSF compiler from the PSF Toolkit.

In this context we should mention the Esprit project GIPE, where the ASF+SDF formalism [Kli91] is being developed. In that formalism an algebraic specification can be integrated with a definition of a context free grammar. This means that both input and output terms as well as equations can be written in a user defined syntax. Such a formalism will make the construction of special tools, such as we have shown in this chapter, superfluous.

CHAPTER 8
CASE STUDY II: THE GKS INPUT MODEL

8.1 INTRODUCTION & MOTIVATION

In this chapter we will apply PSF to a more or less 'real-world' problem. The aim is to give a specification of part of the ISO (International Standards Organization) graphics standard GKS (Graphical Kernel System) [ISO85]. The original ISO specification of the standard is given in a rather informal way and therefore open to ambiguous interpretations.

This case study focuses on the GKS input model, as this is considered one of the more problematic areas in the description of the standard. There are several other papers that deal with this problem notably [DLH89] and [SAHH91]. The first paper uses CSP [Hoa85] as the formal technique to describe the input device, the second uses MANIFOLD [Arb91], a programming language for concurrent communicating systems. The CSP specification from [DLH89] will be the basis for the PSF specifications in this chapter.

In this chapter we will show that although the specification of the GKS input model in [DLH89] is given in a formal description technique, it still contains some minor errors. In our opinion this is partly due to the fact that no computer tools were used to support the specification process. It is our general experience that is quite hard to give error-free specifications of even relatively small problems, if no computer support is used.

To be able to use the PSF Toolkit to study the GKS input model, we have translated the original CSP specification into PSF. Because there are differences between CSP and PSF, this translation is not completely straightforward. The PSF translation tries to stay as close to the original CSP specification as possible.

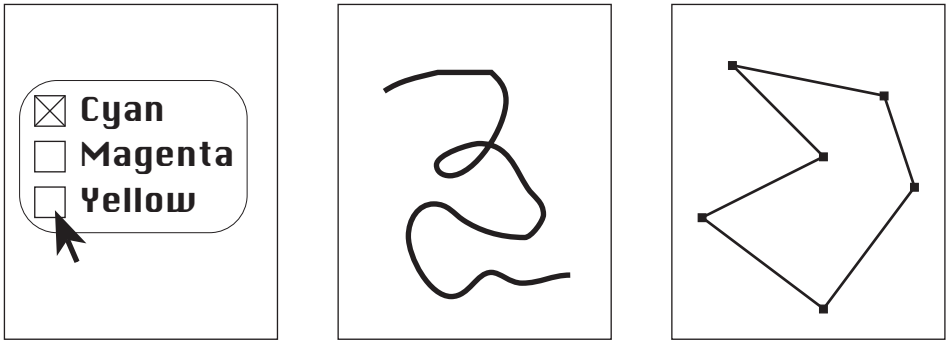
By simulating this PSF specification we gained more insight into the behaviour of the GKS input model and found some errors. In the second part of this chapter we try to correct the problems found, and give a second specification in a more PSF-like style.

8.2 The GKS Input Model

The purpose of the GKS input model is to provide a device-independent interface between an *application program* and several *physical* input devices. In this model the application program does not interact with the physical input devices but with abstractions thereof called *logical* input devices.

There are several classes of logical input devices, which are distinguished by the type of value they return. Some examples of value types are; a point in a coordinate system, a real number or a string of characters. In this chapter we will focus on the *locator device class*, which returns points in a coordinate system. Two examples of physical input devices that belong to this class are a *mouse* and a *tracker ball*.

The GKS input model defines that each logical input device can operate in one of three *operating modes*. These operating modes are called: *request mode*, *sample mode* and *event mode*. Below, we will describe the differences between these operating modes, and relate them to an example of their application in a graphics program.



a. Request Mode *b. Sample Mode* *c. Event Mode*

Figure 8.1 Three different input modes

The *request mode* is the simplest mode. The application asks for a value and waits until this value is available. The value must explicitly be supplied by *triggering* the input device. In case of a *mouse* this would correspond with pressing the *mouse button*. An example of the usage of the request mode is shown in Figure 8.1.a. In this case one point is chosen, and triggered, to indicate the selection of one of three possible choices.

Figure 8.1.b shows the use of the *sample mode*. The application can read the current value of the logical input device at any moment. In this mode the value does not have to be triggered. An example of the application of this operating mode, is the drawing of a *freehand* curve.

The final operating mode is the *event mode*. This mode is related to the sample mode, but instead of the system deciding the moment at which sampling occurs, the user has to supply sampling points explicitly. The values generated by the user are stored in an internal queue. This means that the application does not necessarily have to keep in pace

with the rate of input values, but reads its values from the internal queue at its own pace. Figure 8.1 c shows an example of the usage of the event mode in the definition of a polygon.

8.3 TWO SPECIFICATIONS

In this section we will give two specifications of the GKS input model. We present a specification in CSP taken from [DLH89] and its translation into PSF. The PSF specification given is as *literal* as possible a translation of the original CSP specification. The differences in the specifications are due to the differences between CSP and PSF.

8.3.1 GKS INPUT DEVICE IN CSP

The graphical representation of the different processes of the CSP specification of the GKS input model is given in Figure 8.2. This picture, taken from [DLH89], shows the cooperating processes as boxes and the actions they share in italic typeface.

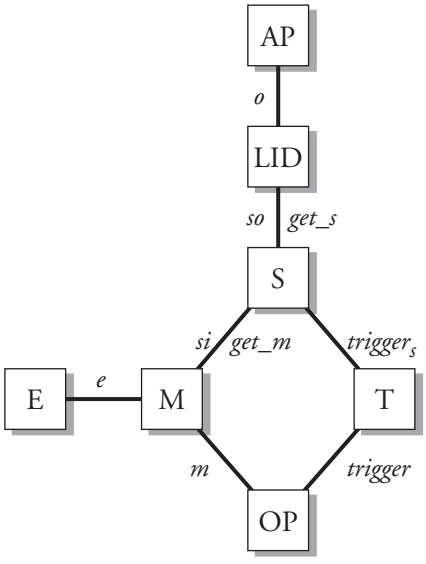


Figure 8.2 Process structure of the CSP specification

We will clarify the abbreviations used for the processes and describe their global behaviour below. For a detailed description of the behaviour of the CSP specification we refer to [DLH89].

- AP : Application
  The application can be put in one of four states, one of the three operating modes and the *quiescent mode* in which it is idling. If it is put in one of the operating modes it starts with activating all other processes. Then it waits until it is able to read a value from channel *o*.

- LID : Logical Input Device
  The logical input device process signals the storage process that it wants a new value by means of a *get_s*. Then it reads a value from channel *so* and sends this value along channel *o* to the application.

- S : Storage
  The storage process is a buffer to store the logical value. In the event mode, storage is provided by means of a queue, in the other cases by a one-bit buffer. The storage process waits for the request of the logical input device first (*get_s*). Then it signals the measure process that it wants a value by means of *get_m*. In case of the request mode and event mode *get_m* is preceded by a *trigger* communication with the trigger process. Subsequently it reads a value from the measure process at channel *si* and sends it along channel *so* to the logical input device process.

- M : Measure
  The task of the measure process is to convert physical values it receives from the operator process through channel *m*, to logical values. After receiving a physical value, the corresponding logical value is sent to the echo process along channel *e*. After a request for the current value from the storage process by means of *get_m*, the logical value is sent along channel *si*.

- OP : Operator
  The operator process generates new physical values that are sent to the measure process through channel *m*. In case of the request mode and event mode the operator can also engage in the action *trigger* with the trigger process.

- T : Trigger
  The trigger process first engages in a *trigger* action with the operator process and then passes on this signal via $trigger_s$ to the storage process.

- E : Echo
  The echo process just reads new logical values from the measure process along channel *e*.

The original CSP specification from [BLH89] contains two minor syntactical errors in the specification of the recursive definitions, which have been corrected here. Apart from this the following specification is a verbatim copy.

$$
\begin{aligned}
&AP = \\
&\quad \text{set-mode-quiescent} \rightarrow AP \\
&\quad \square\ (\text{request} \rightarrow \text{REQUEST} \parallel E_0 \parallel M_0 \parallel T \parallel OP^R \parallel S^R \parallel LID^R) \\
&\quad \square\ (\text{set-mode-sample} \rightarrow \text{SAMPLE} \parallel E_0 \parallel M_0 \parallel OP^S \parallel S^S \parallel LID^S) \\
&\quad \square\ (\text{set-mode-event} \rightarrow \text{EVENT} \parallel E_0 \parallel M_0 \parallel T \parallel OP^E \parallel S^E_{<>} \parallel LID^E)
\end{aligned}
$$

$$
\begin{aligned}
&OP^R = (m!v \rightarrow OP^R) \square (\text{trigger} \rightarrow \text{STOP}_{OP^R}) \\
&\text{REQUEST} = o?v \rightarrow AP \\
&M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \square (\text{get\_m} \rightarrow si!v \rightarrow M_v) \\
&T = \text{trigger} \rightarrow \text{trigger}_s \rightarrow T \\
&E_v = e?v' \rightarrow E_{v'} \\
&LID^R = \text{get\_s} \rightarrow so?v \rightarrow o!v \rightarrow LID^R \\
&S^R = \text{get\_s} \rightarrow \text{trigger}_s \rightarrow \text{get\_m} \rightarrow si?v \rightarrow so!v \rightarrow S^R
\end{aligned}
$$

$OP^S = (m!v \rightarrow OP^S)$
$SAMPLE = \mu X: (sample \rightarrow o?v \rightarrow X) \mid (set\text{-}mode\text{-}quiescent \rightarrow AP)$
$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$
$E_v = e?v' \rightarrow E_{v'}$
$LID^S = sample \rightarrow get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^S$
$S^S = get\_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^S$

$OP^E = (m!v \rightarrow OP^E) \mid (trigger \rightarrow OP^E)$
$EVENT = \mu X: (await\_event \rightarrow o?v \rightarrow X) \mid (set\text{-}mode\text{-}quiescent \rightarrow AP)$
$M_v = (m?v' \rightarrow e!v' \rightarrow M_{v'}) \mid (get\_m \rightarrow si!v \rightarrow M_v)$
$T = trigger \rightarrow trigger_s \rightarrow T$
$E_v = e?v' \rightarrow E_{v'}$
$LID^E = await\_event \rightarrow get\_s \rightarrow so?v \rightarrow o!v \rightarrow LID^E$
$S^E_{S<v>} = get\_s \rightarrow so!v \rightarrow S^E_S$
$S^E_S = trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>S}$
$S^E_{<>} = (get\_s \rightarrow (time\_out \rightarrow so!NONE \rightarrow trigger_s \rightarrow S^E_{<>}$
$\mid trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^E_{<>}))$
$\mid (trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>})$

8.3.2 GKS INPUT DEVICE IN PSF

In this section we will give a translation of the CSP specification into PSF. The PSF specification has been constructed in a modular fashion, and so we will discuss the translation per module. The modular structure of the specification is given in Figure 8.3. The arrows indicate the *import* structure.

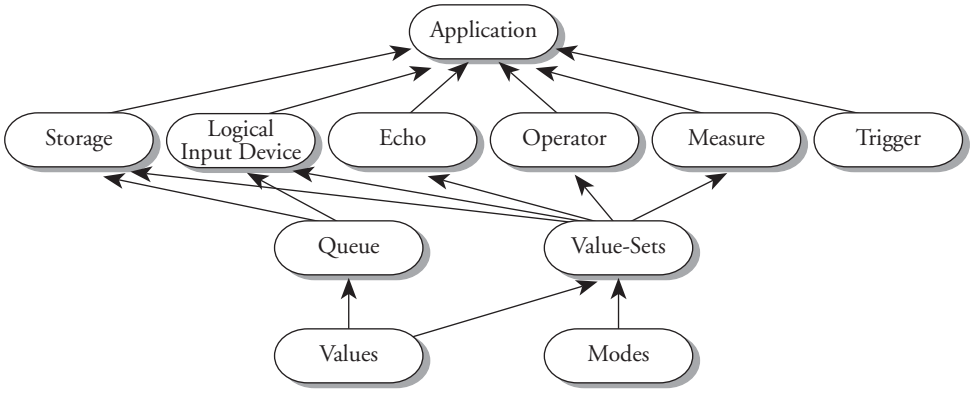


Figure 8.3 Modular structure of the PSF translation

We have to make one important remark concerning the translation. The communication schemes of CSP and PSF differ significantly. Communication in CSP can occur between any number of processes, while communication in PSF is always binary. This causes a problem for a general translation scheme, but luckily the communication defined in the CSP specification at hand is of a binary nature.

A second remark concerns the use of data types in the translation. Because PSF supports the use of algebraically defined data types, we have included them in the PSF translation, following the suggestions given in [DLH89]. Data types are introduced for both the *physical* and *logical values*, for the three different *operation modes* of the GKS input device and the *queue* used in the storage process in event mode.

8.3.3 THE TRANSLATION

8.3.3.1 MODES

The different modes of the GKS input device are modelled by module *Modes*. It specifies a data type consisting of three constants, namely: *R* for request, *S* for sample and *E* for event. These constants will serve as parameters for process descriptions in the specification.

```
data module Modes
begin

  exports
  begin

    sorts
      MODE

    functions
      R : -> MODE
      S : -> MODE
      E : -> MODE

  end

end Modes
```

8.3.3.2 TRIGGER

The translation of the triggering process is straightforward.

```
process module Trigger
begin

  exports
  begin
    atoms
      trigger, trigger-s
    processes
      T
  end

  definitions
    T =
      trigger . trigger-s . T

end Trigger
```

8.3.3.3 VALUES

In the PSF specification we distinguish between two types of values as opposed to the CSP specification from [DLH89], namely: physical values generated by the operator, modelled by elements from the set *PHYSICAL-VALUE* and logical values, which are dealt with by the logical input device and the application, modelled by *LOGICAL-VALUE*.

For testing purposes we have defined four different physical values named *pval-0 … pval-3* and five different logical values named *lval-0 … lval-3* and *NONE*. The constant *NONE* is the special logical value transmitted when an empty queue is sampled. The value *0* occurring in the original specification is equal to *lval-0*. Finally a function *f* is defined, mapping physical values into logical values as suggested in [DLH89].

```
data module Values
begin

  exports
  begin
    sorts
      PHYSICAL-VALUE,
      LOGICAL-VALUE
    functions
      pval-0 : -> PHYSICAL-VALUE            -- physical values
      pval-1 : -> PHYSICAL-VALUE
      pval-2 : -> PHYSICAL-VALUE
      pval-3 : -> PHYSICAL-VALUE
      lval-0 : -> LOGICAL-VALUE             -- logical values
      lval-1 : -> LOGICAL-VALUE
      lval-2 : -> LOGICAL-VALUE
      lval-3 : -> LOGICAL-VALUE
      NONE : -> LOGICAL-VALUE               -- special constants
      0 : -> LOGICAL-VALUE
      f : PHYSICAL-VALUE -> LOGICAL-VALUE   -- mapping function
  end

  equations
    [Mx] 0 = lval-0
    [M0] f(pval-0) = lval-0
    [M1] f(pval-1) = lval-1
    [M2] f(pval-2) = lval-2
    [M3] f(pval-3) = lval-3

end Values
```

To make sure that the simulator recognizes the fact that finite data domains are used, we define sets for the different data domains by enumerating the constant elements.

```
process module Value-Sets
begin

  exports
  begin
    sets
      of PHYSICAL-VALUE
           PV-SET = {pval-0,pval-1,pval-2,pval-3}
      of LOGICAL-VALUE
           LV-SET = {lval-0,lval-1,lval-2,lval-3,NONE}
```

```
        of MODE
            MODE-SET = {R,S,E}
    end

    imports
      Values, Modes

end Value-Sets
```

8.3.3.4 OPERATOR

In the PSF specification we have chosen to group the three different instantiations of the processes, one for each operating mode, into a single module to achieve better modularity. In the following module the operator process is specified.

Note that the CSP input construct: $c?x \rightarrow P(x)$ is translated into the PSF expression *sum(x in X, P(x))*, where *X* is the data domain of variable *x*.

```
process module Operator
begin

  exports
  begin
    atoms
      m : PHYSICAL-VALUE
      trigger
    processes
      OP : MODE
  end

  imports
    Value-Sets

  definitions
    OP(R) = sum(v in PV-SET, m(v)) . OP(R) + trigger
    OP(S) = sum(v in PV-SET, m(v)) . OP(S)
    OP(E) = sum(v in PV-SET, m(v)) . OP(E) + trigger . OP(E)

end Operator
```

In all three operating modes process *OP* is capable of generating new values and sending them along *m*. In the request mode there is an alternative *trigger* action, after which the process terminates, and in the event mode a *trigger* followed by a return to the initial state of *OP*.

8.3.3.5 LOGICAL INPUT DEVICE

Process *LID* asks the storage process for a value by means of *get-s*, then reads this value through *so(v)* and passes it along via *o(v)* before it returns to its initial state. In the sample mode this process is preceded by a *sample* action, and in the event mode by an *await-event* action.

```
process module Logical-Input-Device
begin

  exports
  begin
    atoms
      get-s
      so : LOGICAL-VALUE
      o : LOGICAL-VALUE
      sample
      await-event
    processes
      LID : MODE
  end

  imports
    Value-Sets

  definitions
    LID(R) =
      get-s . sum(v in LV-SET, so(v) . o(v)) . LID(R)
    LID(S) =
      sample . get-s . sum(v in LV-SET, so(v) . o(v)) . LID(S)
    LID(E) =
      await-event . get-s . sum(v in LV-SET, so(v) . o(v)) . LID(E)
end Logical-Input-Device
```

8.3.3.6 MEASURE

The process Measure reads physical values and transforms them into logical values, which are to be transmitted further. Because PSF supports specification of data we have included the translation from physical values into logical values, as suggested in [DLH89], by means of function *f*.

```
process module Measure
begin

  exports
   begin
     atoms
      m : PHYSICAL-VALUE
      e : LOGICAL-VALUE
      si : LOGICAL-VALUE
      get-m
     processes
      M : LOGICAL-VALUE
   end

   imports
    Value-Sets

   variables
    v :-> LOGICAL-VALUE
```

```
    definitions
     M(v) =
       sum(v' in PV-SET, m(v') . e(f(v')) . M(f(v'))) +
       get-m . si(v) . M(v)

end Measure
```

8.3.3.7 ECHO

The translation of the echo process is straightforward.

```
process module Echo
begin
  exports
  begin
    atoms
      e : LOGICAL-VALUE
    processes
      E : LOGICAL-VALUE
  end

  imports
    Value-Sets

  variables
    v : -> LOGICAL-VALUE

  definitions
    E(v) =
      sum(v' in LV-SET, e(v') . E(v'))

end Echo
```

8.3.3.8 QUEUE

In PSF, as opposed to CSP, it is possible to give an algebraic specification of the data types to be used in the process specification. The following specification of a queue is used in the storage process in the event mode. Elements are added to a queue by means of the function *insert()*.

```
data module Queue
begin

  exports
  begin
    sorts
      QUEUE
    functions
      empty-queue : -> QUEUE
      insert : LOGICAL-VALUE # QUEUE -> QUEUE
      qc : LOGICAL-VALUE # QUEUE -> QUEUE       -- queue constructor
  end

  imports
    Values
```

```
    variables
      q : -> QUEUE
      e, e': -> LOGICAL-VALUE

    equations
      [Q1] insert(e,empty-queue) = qc(e,empty-queue)
      [Q2] insert(e,qc(e',q)) = qc(e',insert(e,q))

  end Queue
```

8.3.3.9 STORAGE

The specification of the storage process in PSF employs an algebraically defined queue. The specification of the storage process in the event mode is slightly different from the original CSP specification. In the PSF specification we distinguish between an empty queue and a queue that has at least one element, in the process definition of the storage element in event mode. The original specification is not that precise in defining a queue and thus introduces three process definitions:

- one for a queue containing at least one element, which sends an element
  $S^E_{S<v>} = get\_s \rightarrow so!v \rightarrow S^E_S$
- one for an arbitrary queue, which receives an element
  $S^E_S = trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>S}$
- one for an empty queue, which can either receive or send an element
  $S^E_{<>} = (get\_s \rightarrow (time\_out \rightarrow so!NONE \rightarrow trigger_s \rightarrow S^E_{<>}$
  $\quad | \ trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow so!v \rightarrow S^E_{<>}))$
  $\quad | \ (trigger_s \rightarrow get\_m \rightarrow si?v \rightarrow S^E_{<v>})$

One could argue that there is a conflict between the two latter definitions in the original CSP specification. The queue *S* in the second process definition can be a queue containing an element as well as an empty queue leading to an ambiguous and possibly incorrect interpretation. This problem has been eliminated in the following PSF specification.

```
  process module Storage
  begin

    exports
    begin
      atoms
        si : LOGICAL-VALUE
        so : LOGICAL-VALUE
        get-s
        trigger-s
        get-m
        time-out
      processes
        S : MODE
        S : MODE # QUEUE
    end

    imports
      Value-Sets, Queue
```

```
variables
  q : -> QUEUE
  l : -> LOGICAL-VALUE

definitions
  S(R) =
    get-s . trigger-s . get-m . sum(v in LV-SET, si(v) . so(v)) . S(R)
  S(S) =
    get-s . get-m . sum(v in LV-SET, si(v) . so(v)) . S(S)
  S(E,qc(l,q)) =
    get-s . so(l) . S(E,q) +
    trigger-s . get-m .
    sum(v in LV-SET, si(v) . S(E,insert(v,qc(l,q))))
  S(E,empty-queue) =
    get-s . (
      time-out . so(NONE) . S(E,empty-queue) +
      trigger-s . get-m . sum(v in LV-SET, si(v) . so(v)) .
      S(E,empty-queue)
    ) +
    trigger-s . get-m .
    sum(v in LV-SET, si(v) . S(E,insert(v,empty-queue)))

end Storage
```

8.3.3.10 APPLICATION

Finally, all process modules are imported into module *Application*. The communication scheme has to be defined here.

Because communication between processes in CSP is based on synchronizing actions with identical names, we have to change the names of the atomic actions in the PSF specification slightly. This is done by renaming the actions when the module in which these actions were defined is imported. All names of actions used to send data are preceded by *'s-'* and all names of actions used to receive data are preceded by *'r-'*. The result of the communication between sending and receiving pairs is defined to be the original (CSP) name.

The recursive definitions using the CSP μX: (… $\rightarrow$ X) construct, are translated into PSF using an auxiliary sub-process, labelled with a prime, which takes care of the recursion.

```
process module Application
begin

  exports
  begin
    atoms
      set-mode-quiescent
      request
      set-mode-sample
      set-mode-event
      r-o : LOGICAL-VALUE
      r-sample
      r-await-event
      m : PHYSICAL-VALUE
      si : LOGICAL-VALUE
      so : LOGICAL-VALUE
```

```
    o : LOGICAL-VALUE
    e : LOGICAL-VALUE
    get-m
    get-s
    trigger
    trigger-s
    sample
    await-event
  processes
    System, AP
end

imports
  Value-Sets,

  Operator {
     renamed by [
      m -> s-m,
      trigger -> s-trigger
    ]
  },

  Logical-Input-Device{
     renamed by [
      get-s -> s-get-s,
      so -> r-so,
      o -> s-o,
      sample -> s-sample,
      await-event -> s-await-event
    ]
  },

  Measure{
     renamed by [
      m -> r-m,
      e -> s-e,
      get-m -> s-get-m,
      si -> s-si
    ]
  },

  Echo{
     renamed by [
      e -> r-e
    ]
  },

  Trigger{
     renamed by [
      trigger -> r-trigger,
      trigger-s -> r-trigger-s
    ]
  },
```

```
  Storage{
     renamed by [
      get-s -> r-get-s,
      trigger-s -> s-trigger-s,
      get-m -> r-get-m,
      si -> r-si,
      so -> s-so
    ]
  }

processes
  REQUEST, SAMPLE, SAMPLE', EVENT, EVENT'

sets of atoms
  H = {
    s-m(p), r-m(p),
    s-si(l), r-si(l),
    s-so(l), r-so(l),
    s-o(l), r-o(l),
    s-e(l), r-e(l),
    s-get-m, r-get-m,
    s-get-s, r-get-s,
    s-trigger, r-trigger,
    s-trigger-s, r-trigger-s,
    s-sample, r-sample,
    s-await-event, r-await-event
    | p in PHYSICAL-VALUE, l in LOGICAL-VALUE }

communications
  s-m(p) | r-m(p) = m(p) for p in PHYSICAL-VALUE
  s-si(l) | r-si(l) = si(l) for l in LOGICAL-VALUE
  s-so(l) | r-so(l) = so(l) for l in LOGICAL-VALUE
  s-o(l) | r-o(l) = o(l) for l in LOGICAL-VALUE
  s-e(l) | r-e(l) = e(l) for l in LOGICAL-VALUE
  s-get-m | r-get-m = get-m
  s-get-s | r-get-s = get-s
  s-trigger | r-trigger = trigger
  s-trigger-s | r-trigger-s = trigger-s
  s-sample | r-sample = sample
  s-await-event | r-await-event = await-event

definitions
  System =
    encaps(H, AP)

  AP =
    set-mode-quiescent . AP +
    request .
      (REQUEST || E(0) || M(0) || T || OP(R) || S(R) || LID(R)) +
    set-mode-sample .
      (SAMPLE || E(0) || M(0) || OP(S) || S(S) || LID(S)) +
    set-mode-event .
      (EVENT || E(0) || M(0) || T || OP(E) || S(E,empty-queue) ||
       LID(E))

  REQUEST =
    sum(v in LV-SET, r-o(v)) . AP
```

```
    SAMPLE =
      SAMPLE' + set-mode-quiescent . AP

    SAMPLE' =
      r-sample . sum(v in LV-SET, r-o(v)) . SAMPLE'

    EVENT =
      EVENT' + set-mode-quiescent . AP

    EVENT' =
      r-await-event . sum(v in LV-SET, r-o(v)) . EVENT'

end Application
```

8.4 REMARKS ON THE SPECIFICATION

In this section we will give some remarks concerning the original CSP specification. The first set of remarks deals with the style of specification.

We think it is strange that the specification of the *operator* process depends on the mode of the *logical input device*. We would like to think of these processes as two separate entities, where the specification of the operator process is independent of the input mode of the logical input device.

The specification uses a set of actions to establish proper communication. It is our opinion that this set can be reduced due to fact that a communication cannot take place when one of the partners is not willing to communicate. We think we may leave out the following actions: *get_s*, *get_m*, *trigger_s*.

The different processes are initialized with their default values each time a different input mode is chosen. We would rather see this initialization happen just once. If we would take a *mouse* as example for an input device, the current specification would suggest that the *pointer* of the mouse is reset to its initial position after each mode change, which is of course highly undesirable.

The operator process *OP* has to generate physical values. It is specified to send a value along channel *m* by *m!v*. The variable *v* is not bound in the CSP specification and so in the translation we had to introduce a *sum* construct generating values.

A second remark has to do with the specification of the *SAMPLE* and *EVENT* processes:

SAMPLE = μX: (sample → o?v → X) | (set-mode-quiescent → AP)

EVENT = μX: (await_event → o?v → X) | (set-mode-quiescent → AP)

Using the simulator from the PSF Toolkit we found out that once the *sample* or *await_event* action is chosen, these processes will never offer *set-mode-quiescent* again, thus resulting in a *livelock*.

The third remark deals with termination of sub-processes of *AP* and has also been discovered by simulating the specification. To illustrate this problem we will first give the trace of a possible execution. The entries in normal type face are process terms describing the current *state* of the process. The entries in bold face are the atomic actions that are executed. The sub-terms that are responsible for the next step are given in italic type face.

```
System
  request
encaps(H,(REQUEST || E(0) || M(0) || T || OP(R) || S(R) || LID(R)))
  trigger
encaps(H,(REQUEST || E(0) || M(0) || trigger-s . T || S(R) || LID(R)))
  get_s
encaps(H,(REQUEST || E(0) || M(0) || trigger-s . T || trigger-s . get-m . sum(v
in LV-SET, si(v) . so(v)) . S(R) || sum(v in LV-SET, so(v) . o(v)) . LID(R)))
  trigger-s
encaps(H,(REQUEST || E(0) || M(0) || T || get-m . sum(v in LV-SET, si(v) .
so(v)) . S(R) || sum(v in LV-SET, so(v) . o(v)) . LID(R)))
  get_m
encaps(H,(REQUEST || E(0) || si(v) . M(v) || T || sum(v in LV-SET, si(v) .
so(v)) . S(R) || sum(v in LV-SET, so(v) . o(v)) . LID(R)))
  si(0)
encaps(H,(REQUEST || E(0) || M(0) || T || so(0) . S(R) || sum(v in LV-SET,
so(v) . o(v)) . LID(R)))
  so(0)
encaps(H,(REQUEST || E(0) || M(0) || T || S(R) ||  o(0) . LID(R)))
  o(0)
encaps(H,(AP || E(0) || M(0) || T || S(R) || LID(R)))
  request
encaps(H,(REQUEST || E(0) || M(0) || T || OP(R) || S(R) || LID(R) || E(0) ||
M(0) || T || S(R) || LID(R)))
```

This trace shows that after selecting the *request mode* and after transmitting a value to the application over channel *o*, the process reaches a state where *AP* is 'called' recursively, but where not all its sub-processes have become 'inactive'. This has as a result that after the given trace there are two different instances of the *measure*, *echo*, *trigger*, *storage*, and *logical input device* processes. Admittedly, using the equality $X \parallel X = X$, which is valid in CSP, two equal processes in parallel can be reduced to one instance of that process. However, we think that this is nevertheless an unintended feature of the specification.

8.5 A CORRECTED PSF SPECIFICATION

In this section we want to redo the specification of the GKS input model in PSF. This means that we give the specification now in a more PSF-like style. Processes and atoms get longer identifier names and the parallelism between objects has been regrouped. Moreover we have tried to correct the problems we have found in the original specification.

Figure 8.4 shows the relation between the different processes in the new specification. The operator process still generates physical values, which are passed to the logical input device. Moreover it generates *trigger* signals in this specification.

The logical input device transforms the physical values into logical values and passes them on to the echo process as well as the application.

The application still reads logical values. It can also put the logical input device into the three different modes by means of a *mode-switch*, and reset the logical input device by means of a *reset* action.

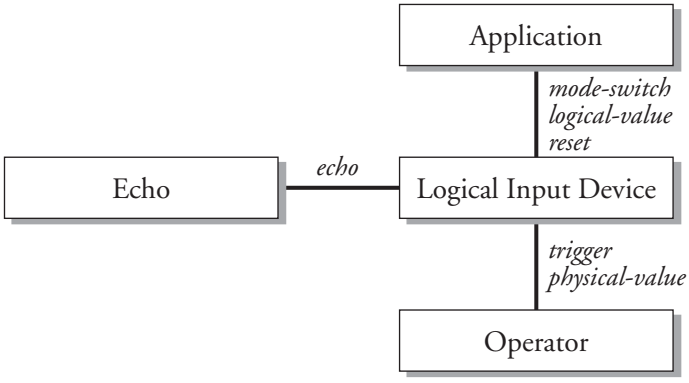


Figure 8.4 Process structure of the new PSF specification

The following figure shows the modular structure of the new PSF specification. It is quite similar to the structure of the direct translation of the CSP specification, except for *Measure*, *Trigger* and *Storage*, which now have been integrated in the logical input device and operator processes.

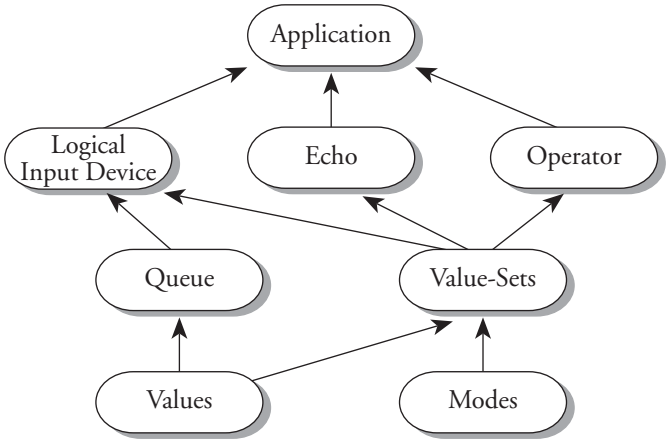


Figure 8.5 Modular structure of the new PSF specification

The following listing defines the new PSF specification of the GKS input device.

8.5.1 GKS DATA SPECIFICATION

The data modules have not changed significantly in the new specification. The identifier names are more descriptive and the constant value *0* has been discarded. Module *Queues* has not changed at all and therefore it is not included here.

```
data module Values
begin

  exports
  begin
    sorts
     PHYSICAL-VALUE,
     LOGICAL-VALUE
    functions
     pval-0 : -> PHYSICAL-VALUE    -- physical values
     pval-1 : -> PHYSICAL-VALUE
     pval-2 : -> PHYSICAL-VALUE
     pval-3 : -> PHYSICAL-VALUE
     lval-0 : -> LOGICAL-VALUE     -- logical values
     lval-1 : -> LOGICAL-VALUE
     lval-2 : -> LOGICAL-VALUE
     lval-3 : -> LOGICAL-VALUE
     lval-NONE : -> LOGICAL-VALUE
     pl-map : PHYSICAL-VALUE -> LOGICAL-VALUE    -- mapping function
  end

  equations
    [M0] pl-map(pval-0) = lval-0
    [M1] pl-map(pval-1) = lval-1
    [M2] pl-map(pval-2) = lval-2
    [M3] pl-map(pval-3) = lval-3

end Values
```

Module *Modes* describes the three different operating modes.

```
data module Modes
begin

  exports
  begin
    sorts
     MODE
    functions
     request-mode : -> MODE
     sample-mode : -> MODE
     event-mode : -> MODE
  end

end Modes
```

The data values are again grouped into sets to make sure that the PSF Toolkit can recognize that finite data types are used.

```
process module Value-Sets
begin

  exports
  begin
    sets
     of PHYSICAL-VALUE
       PV-SET = {pval-0,pval-1,pval-2,pval-3}
```

```
      of LOGICAL-VALUE
        LV-SET = {lval-0,lval-1,lval-2,lval-3,lval-NONE}
      of MODE
        MODE-SET = {request-mode, sample-mode, event-mode}
  end

  imports
    Values, Modes

end Value-Sets
```

8.5.2 THE OPERATOR

The operator has changed such that it no longer depends on the input mode of the logical input device. The process consists of two sub-processes. One process generates new physical values constantly, modelled by action *new-physical-value*. In a mouse device this would be the ball and accompanying hardware registering the movement of the mouse.

The other process is responsible for the delivery of physical values. It always offers the possibility to communicate the most recent new physical value through *send-physical-value*. Furthermore, it offers the possibility to send a trigger, modelled by the action *send-trigger*, which is immediately followed by a *send-physical-value*. In a mouse device, for example, this corresponds with pushing the mouse button.

```
process module Operator
begin

  exports
  begin
    atoms
      send-physical-value : PHYSICAL-VALUE
      new-physical-value : PHYSICAL-VALUE
      send-trigger
    processes
      Operator
  end

  imports
    Value-Sets

  processes
    New-Value
    Operator : PHYSICAL-VALUE

  variables
    pv : -> PHYSICAL-VALUE

  definitions
    Operator =
      Operator(pval-0)      -- initialization

    New-Value =
      sum(v in PV-SET, new-physical-value(v) . Operator(v))
```

```
    Operator(pv) =
      New-Value +
      send-physical-value(pv) . Operator(pv) +
      send-trigger . send-physical-value(pv) . Operator(pv)

end Operator
```

8.5.3 THE LOGICAL INPUT DEVICE

The specification of the *measure* and *storage* process is now embedded in specification of the logical input device. In our opinion modelling these processes separately is superfluous.

The *LID* process (without parameters) is responsible for distributing the request for the different operating modes. The three modes are specified by the *LID : MODE* process. The process specifying the behaviour of the logical input device in the event mode consists of two sub-processes. One process receives values from the *operator* and adds these values to the internal queue. The other process is responsible for retrieving values from the queue and delivering them to the application.

```
process module Logical-Input-Device
begin

  exports
  begin
    atoms
      read-mode-switch : MODE
      read-physical-value : PHYSICAL-VALUE
      send-echo : LOGICAL-VALUE
      send-logical-value : LOGICAL-VALUE
      read-trigger
      read-reset
      time-out
    processes
      LID
  end

  imports
    Value-Sets, Queue

  processes
    LID : MODE
    Sample-LID : LOGICAL-VALUE
    Q-LID : QUEUE
    Q-write : QUEUE

  variables
    q : -> QUEUE
    l : -> LOGICAL-VALUE

  definitions
    LID =
       sum(m in MODE-SET, read-mode-switch(m) . LID(m))
```

```
    LID(request-mode) =
      read-trigger .
      sum(pv in PV-SET,
        read-physical-value(pv) .
        send-echo(pl-map(pv)) .
        send-logical-value(pl-map(pv))
      ) . LID

    LID(sample-mode) =
      sum(pv in PV-SET,
        read-physical-value(pv) .
        send-echo(pl-map(pv)) .
        Sample-LID(pl-map(pv))
      ) +
      read-reset .
      LID

    Sample-LID(l) =
      send-logical-value(l) . Sample-LID(l) +
      LID(sample-mode)

    LID(event-mode) =
      Q-LID(empty-queue)

    Q-LID(q) =
      read-trigger .
      sum(pv in PV-SET,
        read-physical-value(pv) .
        send-echo(pl-map(pv)) .
        Q-LID(insert(pl-map(pv),q))
      ) +
      Q-write(q) +
      read-reset .
      LID

    Q-write(qc(l,q)) =
      send-logical-value(l) .
      Q-LID(q)

    Q-write(empty-queue) =
      time-out .
      send-logical-value(lval-NONE) .
      Q-LID(empty-queue)

end Logical-Input-Device
```

8.5.4 THE ECHO PROCESS

The specification of the *echo* process has not changed except for the identifier names and the fact that echoing is now explicitly modelled by the action *echo-value*.

```
process module Echo
begin

  exports
  begin
    atoms
      echo-value : LOGICAL-VALUE
      read-echo : LOGICAL-VALUE
    processes
      Echo
  end

  imports
    Value-Sets

  variables
    v : -> LOGICAL-VALUE

  definitions
    Echo = sum(v in LV-SET, read-echo(v) . echo-value(v)) . Echo

end Echo
```

8.5.5 THE APPLICATION

Finally all modules are combined in the module *Application*. We have added an explicit *reset* action to the specification. Before the application wants to switch to another mode it has to execute the action *send-reset*.

```
process module Application
begin

  exports
  begin
    atoms
      physical-value : PHYSICAL-VALUE
      logical-value : LOGICAL-VALUE
      echo : LOGICAL-VALUE
      mode-switch : MODE
      reset
      trigger
  end

  imports
    Operator, Logical-Input-Device, Echo

  atoms
    read-logical-value : LOGICAL-VALUE
    send-mode-switch : MODE
    send-reset
    quiescent

  processes
    System, Application, Request, Sample, Event
```

```
  sets of atoms
    H = { send-physical-value(p), read-physical-value(p),
          send-logical-value(l), read-logical-value(l),
          send-echo(l), read-echo(l),
          send-mode-switch(m), read-mode-switch(m),
          send-reset, read-reset,
          send-trigger, read-trigger
          | p in PHYSICAL-VALUE, l in LOGICAL-VALUE, m in MODE }
    I = { physical-value(p), echo(l), trigger
          | p in PHYSICAL-VALUE, l in LOGICAL-VALUE }

  communications
    send-physical-value(p) | read-physical-value(p) =
      physical-value(p) for p in PHYSICAL-VALUE
    send-logical-value(l) | read-logical-value(l) =
      logical-value(l) for l in LOGICAL-VALUE
    send-echo(l) | read-echo(l) =
      echo(l) for l in LOGICAL-VALUE
    send-mode-switch(m) | read-mode-switch(m) =
      mode-switch(m) for m in MODE
    send-reset | read-reset = reset
    send-trigger | read-trigger = trigger

  definitions

    System =
      hide(I,encaps(H, Application || Operator || LID || Echo ))
    Application =
      ( quiescent +
        send-mode-switch(request-mode) . Request +
        send-mode-switch(sample-mode) . Sample +
        send-mode-switch(event-mode) . Event ) . Application
    Request =
      sum(v in LV-SET, read-logical-value(v))
    Sample =
      sum(v in LV-SET, read-logical-value(v)) . Sample + send-reset
    Event =
      sum(v in LV-SET, read-logical-value(v)) . Event + send-reset

end Application
```

8.6 CONCLUSIONS

One of the main conclusions of this case study is that it is quite hard to give a correct specification in a formal description formalism of the problem at hand, without the use of tools. The syntax checker and type checker of the PSF Tookit showed their benefit in the initial phase of the specification process.

After we passed this first stage, the PSF simulator was very useful to find unintended execution paths and unexpected *deadlocks* and *livelocks*. It is our opinion that the mistakes in the original CSP specification would have easily been found, if the specification would have been simulated.

A second conclusion is that the size of a specification grows considerably when transforming it from a non-executable formal description into an executable one. In this case study the original CSP specification was 34 lines long. The PSF version of this specification is twelve times as long. From this PSF specification roughly 25% of the code is dedicated to the specification of the data types that are not included in the CSP version, and another 25% of the code is used to carry out the special renaming scheme for the communications. The second PSF specification is nine times as long as the CSP specification. Of this code approximately 30% is used for the specification of data types.

There are several reasons for this phenomenon of growth of code. The most important reason is that the executable specification has to be more exact. Although the CSP specification studied in [DLH89] is a formal specification, there are things that are left unspecified like the queue, for example. Leaving such things out in an executable specification is impossible because we have to supply a simulator with all the information and cannot expect that it '*understands*' the notion of a queue, without being given an exact specification of the behaviour of a queue. The increase in exactness and length of the specification is most clearly seen in the specification of data types.

A second reason that the specification text becomes larger is the fact that we are dealing with modular specifications. We have to record information about the visibility of objects to the outside world, and the availability of objects from other modules. Specifications could be given without the use of modularization techniques, but this would make the explanation of their behaviour more difficult, and would hamper reuse of specifications.

Another reason for the increase in size is that we sometimes have to please the tools. In our case the specification contains an intermediate module *Value-Sets* explicitly defining finite sets of data elements, because the simulator cannot handle general, possibly infinite, sorts as parameter in the *sum* construct.

A final reason is the style of the specification document. Because one has to define more different entities in an executable specification, for example: actions and processes have to be *declared* before their usage, it is wise to adopt a certain layout to emphasize grouping of objects. This introduces empty lines, that only serve as a means to increase the legibility of the text. Moreover, we have given the process definitions in a style where for more complex processes the understandability is enhanced by suggesting the structure using indentation.

In this respect we would like to mention that the figures we have given above also suggest that style plays a role when we compare the two PSF specifications. The second PSF specification, in which we have tried to adhere to a more PSF-like style, is 75% of the size of the first specification in which have tried to give a more direct translation of the original CSP specification. Moreover, the second PSF specification is easier to understand.

A final conclusion confirmed by this case study is that whatever formal method is used, there is no such thing as the ultimate specification of a problem. Differences are the result of using different formal description techniques, but are also heavily influenced by the level of abstraction chosen in modelling a problem. In our case this is illustrated by the introduction of data types in the PSF specifications, and by the differences between the two PSF specifications caused by regrouping of the parallel processes.

Finally we would like to state that during the process of specification one has to make a number of decisions that are (currently) only taken on basis of one's personal taste. This strongly suggests that some of the problems that normally are associated with traditional programming languages will not be solved by the use of formal description techniques.

CHAPTER 9
DISCUSSION, COMPARISON & FUTURE DEVELOPMENTS

9.1 INTRODUCTION

In the previous chapters we have described the specification language PSF and its intermediate languages as well as the set of programs that make up the PSF Toolkit. In this final chapter we will discuss some issues that are not directly related with the implementation of the Toolkit.

There have been developments in research in specification languages that are related to PSF. In this chapter we will discuss two of these languages μCRL [GP90] and XP [Vel92], show the differences with PSF and explain how they are interfaced with the PSF Toolkit. Moreover, we will shortly describe some tool sets that have been developed for other specification languages.

We will also give some attention to the growing use of PSF in different projects. Moreover, we will show some of the problems encountered with the PSF language as well as its implementation and discuss possible future developments relating to PSF and the PSF Toolkit.

9.2 THE LANGUAGE μCRL

The specification language μCRL has been developed as a result of research that has been carried out within the SPECS project, RACE ref. 1046 [SPECS89]. The aim of this project, carried out by a consortium of industrial partners, is to provide a set of computer tools for specification languages, among which SDL [CCITT88] and LOTOS [ISO89a], which can be translated into and combined by way of an intermediate language called CRL (Common Representation Language) [SPECS89]. To be able to better study the basics of a specification language μCRL was developed using the essential elements from CRL.

9.2.1 MOTIVATION

The language μCRL is defined in [GP90] in which it is stated that most specification languages are optimized with respect to usability and that as a result they turn out to be quite complicated, especially when semantics is concerned. Therefore, μCRL aims at being a specification language consisting of core constructs only, but with a thoroughly defined semantics.

The authors express the hope that extensive study of the basic constructs of a specification language by means of μCRL will yield fundamental insights that are hard to obtain through other '*larger*' specification languages. Because of the *'smallness'* of μCRL, they expect that it will not be suited so well as an actual specification language, but more as a base for building tools.

9.2.2 SYNTAX

The syntax of μCRL has much in common with PSF. As in PSF the various objects are defined in blocks, which are preceded by a keyword. In μCRL there is no specific order in which these blocks have to be defined and blocks of the same type may appear more than once in specification. To give an impression of a μCRL specification we will repeat the example of an alternating bit protocol given in [GP90].

```
sort    Bool
func    T,F:->Bool

sort    D
func    d1,d2,d3 : -> D

sort    error
func    e :-> error

sort    bit
func    0,1 : -> bit
        invert : bit -> bit

rew     invert(1)=0
        invert(0)=1

act     r1,s4           : D
        s2,r2,c2        : D # bit
        s3,r3,c3        : D # bit
        s3,r3,c3        : error
        s5,r5,c5        : bit
        s6,r6,c6        : bit
        s6,r6,c6        : error

comm    r2|s2 = c2
        r3|s3 = c3
        r5|s5 = c5
        r6|s6 = c6
```

```
proc    S               = S(0).S(1).S
        S(n:bit)        = sum(d:D,r1(d).S(d,n))
        S(d:D,n:bit)    = s2(d,n).(r6(invert(n))+r6(e)).S(d,n)+r6(n)

        R               = R(1).R(0).R
        R(n:bit)        = (sum(d:D,r3(d,n))+r3(e)).s5(n).R(n)+
                           sum(d:D,r3(d,invert(n)).s4(d).s5(invert(n)))

        K               = sum(d:D,sum(n:bit,r2(d,n).(tau.s3(d,n)+
                          tau.s3(e)))).K
        L               = sum(n:bit,r5(n).(tau.s6(n)+tau.s6(e))).L

        ABP             = hide({c2,c3,c5,c6},
                            encap({r2,r3,r5,r6,s2,s3,s5,s6}, S||R||K||L))
```

9.2.3 COMPARISON WITH PSF

Some differences with PSF that are striking at first sight are that μCRL does without the *begin-end* pairs and that the type of a variable, used as a parameter of a process or used in a *sum* construct, is defined at the place where the variable is used and not in a separate *variables* section. Because μCRL has no equivalent of the PSF *sets*, the elements of a set used in an encapsulation or abstraction are simply enumerated at the place of usage. As a result of this μCRL specifications are more compact.

Another difference is that μCRL only allows variables to occur at the left-hand side of process definitions instead of data terms as in PSF. This simplifies the implementation of tools and eliminates the possibility of writing specifcations in the following style:

```
X(true) = a
X(not(false)) = b
```

The semantics of PSF has to state explicitly that this specification is equal to:

```
X(true) = a + b
```

Being a language that focuses on semantical rigidity and not user comfort, μCRL lacks any notion of modules. This is of course of no concern for a language that serves as a base for tooling. There are however two features that in our opinion reduce the usability of μCRL as such.

The first problem is the definition of the communication function, which allows only atomic actions with identical input types to communicate. Moreover, the definition of the communication function only lists the names of atomic actions, which implies that a communication definition $a \mid b = c$ applies to all actions with names a and b, irrespective of their input types. This can lead to unexpected side-effects when large specifications were to be written in μCRL or are translated from another language into μCRL.

The second problem originates from the form of the conditional expression in μCRL. This expression consists of two process expressions and a boolean term. Depending on the value of the boolean term one of the two process expressions is chosen. This implies that every μCRL specification must contain at least a boolean sort *Bool* and two truth values: T,F. We think this is quite restrictive for any specification language and even unwanted for a language that is to be used as a base for tooling purposes.

Although μCRL is a core language it contains some operators that are not included in PSF or similar languages. The first two operators are the *left merge* and the *communication merge*. In PSF these operators were left out deliberately because it was reasoned that no one would use these operators in a specification. Although this might be true for the initial specification, in dealing with internal representations used by tools such as the simulator and the proof assistant in the PSF Toolkit the absence of a standard notation for these operators was considered an omission.

Apart from these derived operators, μCRL contains one completely new operator: the *rename* operator. This operator is related to the encapsulation and abstraction operator and renames the names of atomic actions into atomic actions other than δ or τ. It should not be confused with a renaming operator used for renaming imported objects in the setting of a modular specification or programming language. The presence of the *rename* operator in μCRL practically makes the encapsulation and abstraction operators superfluous. By allowing δ and τ to appear in the renaming list, three operators (*encap*, *hide*, *rename*) would be merged into one, leading to a more compact language.

A final problem with the sets specified in the three operators mentioned above is that renaming is performed on basis of the name of the action only (not its type) which may lead to unexpected side effects as sketched earlier in the discussion on the definition of the communication function.

9.2.4 IMPLEMENTATION

There exist two implementations of μCRL. The first implementation is described in [Ver92b] and was written using the ASF+SDF system. The ASF+SDF environment [Kli91] is used to describe the static and dynamic semantics of programming languages. The second implementation provides a translation from μCRL into TIL thereby creating an interface to the PSF Toolkit.

The ASF+SDF implementation is in fact a simulator for μCRL. The static semantics of μCRL is described in the syntax definition formalism SDF [HHKR89] and ASF [BHK89] is used to describe the dynamic semantics of μCRL as well as some simulator specific functionality. Using this definition of μCRL the ASF+SDF environment automatically provides a syntactical editor through the GSE program [Koo94]. In order to provide an interactive user interface to the simulator some functions had to be implemented in LISP directly.

The second implementation is part of the PSF Toolkit and called *mcrl*. It is a more or less traditional transformer translating μCRL into TIL implemented using the *Lex* and *Yacc* packages. Through this translation the full functionality of the PSF Toolkit becomes available for μCRL specifications. There is of course one drawback due to differences between μCRL and PSF as described earlier. The three operators not available in TIL cannot be interpreted correctly but are translated into the following pseudo-TIL operators:

left merge	<lmrg,2> (PE_1 PE_2)
communication merge	<cmrg,2> (PE_1 PE_2)
renaming	<rename>([5.x] *PE*)

Due to the differences in the implementation, the first implementation is largely interpreted (*LISP*, *ASF*) whereas the second is compiled (*C*), the performance of the second implementation regarding execution speed is better.

9.3 THE LANGUAGE XP

The experiences gathered in the design and implementation of PSF have lead to new insights in the design of formal description techniques. To be able to discuss some problems encountered in PSF and to propose alternative solutions we have developed the specification language XP [Vel92]. The name of the language refers to fact that it is an e*xp*eriment in the design of formal description techniques.

9.3.1 MOTIVATION

Some of the strengths and weaknesses of a specification language are only discovered through its use. By implementing the PSF Toolkit and using PSF we have encountered several problems, which we have tried to solve in the design of XP.

One of the design issues of PSF has been that the language should relate as closely as possible to the existing ACP theory and its notation. Although this lead to an easier initial acceptance of the language, we think that in some parts the PSF syntax is unnecessary complex because of this ACP-like notation and therefore difficult to implement and learn for users. Moreover, we have experienced that a number of problems were only encountered in larger specifications, for which the ACP-inspired syntax is not the most optimal. Another problem area is modularization. PSF uses the modularization concept of ASF, which has shown to be one of the weak spots through the years of usage.

One of the main targets of the design of XP has been to try to aim at a larger orthogonality between language constructs in order to keep the language more consistent and compact. This has as result that the language is easier to learn. Another important target has been to try and experiment with a different kind of modularization that tries to minimize dependencies between modules and reduces their interfaces.

9.3.2 BASIC LANGUAGE CONSTRUCTS

As we started the design of XP we realized that it would not be possible to extend PSF in such a way that the extensions would cover the problems we mentioned in the previous section. It was decided that a complete redesign was needed and therefore we have asked ourselves one important question: *What are the basic things we are specifying in a specification language*?

The answer to this question is: *mathematical objects*! We consider four such basic objects, which are divided into two categories. The data objects are *sorts* and *functions*, the process objects are *atomic actions* and *processes*. Other elements of a specification language such as a communication function or equations specifying data types are considered to be *attributes* of one of the four objects.

9.3.2.1 DATA OBJECTS

In XP there is a strict difference between the *declaration* and the *definition* of an object. The *declaration* states the class (sort, function, atom, process) to which an object belongs, its name and possible input and output type. As an example we give the declaration of the sort *Boolean* and the boolean function *and*:

```
sort   Boolean;
func   and(Boolean,Boolean): Boolean;
```

The declaration is preceded by a keyword (*sort*, *func*) and closed by a semi-colon. The semi-colon in XP always functions as a logical *end-of-line*. As in traditional programming languages, this feature greatly facilitates the synchronization of the parser during error recovery. Error recovery in the PSF parser is very crude due to the lack of a comparable separator. If a series of objects of the same type are introduced the keyword does not have to be repeated.

```
func   and(Boolean,Boolean): Boolean;
       or(Boolean,Boolean): Boolean;
       not(Boolean): Boolean;
```

A declaration is turned into a *definition* by supplying the object's attributes. These attributes are inserted before the semi-colon and are enclosed in curly brackets ('{', '}'). As an example we give the definition of the boolean function *and*:

```
func   and(Boolean,Boolean): Boolean {
       for x: Boolean;
         and(true(),x) = x;
         and(false(),x) = false();
       };
```

In XP it is obligatory to specify all rewrite rules defining a function as an attribute of the function. This forces grouping of equations and guarantees that the function is fully defined by these equations. In PSF it is possible to extend a function with an equation even in a different module, which can lead to unexpected side-effects. Variables serving as placeholders in the equations are introduced by the *for* keyword and their scope is limited to the equations in one definition.

9.3.2.2 PROCESS OBJECTS

The three objects in XP that can be parameterized (*functions*, *atoms*, *processes*) are distinguished by the brackets enclosing the parameter list. The three possible forms for a parameter list are:

- functions (…)
- atoms <…>
- processes […]

This implies that each of the four objects has its own name space. Adhering to this convention also implies that empty parameter lists cannot be dropped in a specification. Although this might seem a nuisance at first, we think it helps in understanding larger

specifications. It is certainly a welcome aid for the type checking system as it prevents some of the name clashes that are possible in PSF:

- variables can clash with constant functions in data terms: `and(true,x)`
- atoms can clash with processes in process definitions: `X = a.Z.X`

In the following example, taken from a specification of an alternating bit protocol, we will see the different parameter lists in the definition of the two process objects, atomic actions and processes.

```
atom receive_data<Data> {};
proc RM[Boolean] {
     for b:Boolean;
     RM[b] =
       alt {
       for d:Data;
         seq {
           receive_data<d>;
           SF[b,d];
         }
       }
     };
```

This example shows that process definitions look similar to the definition of the equations from the previous section. It also shows that in XP the main process operators (*sequential*, *alternative*, *parallel composition*) have a prefix notation and that they can have any arity (≥ 1). The choice for this prefix notation was influenced by the programming language Occam [INMOS88] and leads to clearer indentation standards. The three process operators in XP are:

- *seq*, sequential composition.
  seq { x_1; x_2; …; x_n; } is the process that first executes x_1, after termination of x_1 continues with x_2 and so on until it reaches x_n on which termination, the complete expression terminates.
- *alt*, alternative composition.
  alt { x_1; x_2; …; x_n; } is the process that makes a choice between the initial actions of summands x_1 up to x_n, and then proceeds with the execution of the chosen summand. An initial deadlock action is not chosen as long as there are (executable) alternatives.
- *par*, parallel composition.
  par { x_1; x_2; …; x_n; } is the process that represents the simultaneous execution of all processes x_1 up to x_n,

The alternative composition and the parallel composition can be parameterized by variables (generalization) to form the equivalent of the *sum* and *merge* operators from PSF. The generalized commands are formed by inserting the *for* keyword and a list of variables after the operator as shown in the alternative composition in the example above.
If an atom can be the result of the communication between two other atoms this is recorded as an attribute in its definition in the following way:

```
atom data_transferred<Data> {
      for d: Data;
        data_transferred<d> = transmit_data<d> | receive_data<d>;
      };
```

We have experienced that already relatively small process expressions in PSF can become hard to read and understand when the process structure has a high branching degree (extensive use of alternative and parallel composition) and when larger names are used. In understanding such expressions a large amount of time is spent finding matching parentheses. We believe that using the syntax proposed for XP we can more or less overcome this traditional problem due to the use of different brackets, prefix notation and clearer standard indentation.

9.3.3 MODULARIZATION

An XP specification consists of a series of modules. A module in turn is a series of declarations or definitions of objects enclosed within the keyword *module* followed by a module name and the keyword *end*. As opposed to PSF, the order in which the objects are introduced is free, the only restriction is that all objects are declared before they are used. This ensures that the parser is able to fully identify an object at any time.

9.3.3.1 IMPORT

Import of objects from another module is based on importing a complete module in most languages, including PSF. In XP however, it is only possible to import objects from a module one at a time. The reason for this requirement is that we want to be able to type check an XP specification on a modular basis. This implies that all objects that are used by a module *M*, including imported ones, have to be declared within *M*. Along with the declaration the module of origin is specified. The following example shows an import:

```
sort Boolean            { <- Booleans };
func f(): Boolean       { <- Booleans.false };
```

Import of an object is indicated by the backwards arrow: '<-'. It is possible to rename an object on import as shown in the *func* section by means of attaching the original name of the object to the name of the module by means of a dot. The function locally known as *f* is in fact function *false* from module *Booleans*. Besides the possibility to type check on a modular basis, this approach also ensures that the name space of a module is not polluted by the names of all unwanted objects from the imported module.

The situation in ASF and therefore PSF is even worse. Every object that has been imported once, is automatically exported from the importing module. There is no way to shield such objects from further exports.

9.3.3.2 EXPORT

In XP there are two operators that control the visibility of an object for the outside world. The following example illustrates the use of these operators:

```
atom receive_data : <Data>    {}              ->;
func true() : Boolean         { <- Booleans }  -|;
```

The first operator ('->') is the *export* operator, which indicates that the object is visible to the outside world. The second operator ('-|') is the *hiding* operator, which shields an object from the outside world. Attaching one of both operators to all definitions would be a tiresome job. Therefore a default visibility has been defined which is applied in case no export or hiding operator is given. Based on studies of several medium-sized PSF specifications we came to the conclusion that the default visibility control is different for data and process objects. The default visibility operator in XP is *export* for data objects and *hiding* for process objects.

9.3.3.3 COMMUNICATION

In this section we will describe how communication between processes is handled in XP. As opposed to PSF where the communication function is a global property, in XP the communication is defined on a modular level. To support this concept the atomic actions within each module in XP are divided into three classes:

- *inert atoms*:
  actions that do not engage in any communication
- *communicating atoms*:
  actions that form the 'components' of a communication
- *result atoms*:
  actions that are the result of a communication

As in PSF we put some restrictions on the communication schemes that can be defined. The *firm handshaking* restriction from PSF carries over to XP. This restriction implies that a *result atom* cannot take part in another communication. However, because of the modular communication function an exported *result atom* can be a *communicating atom* in the importing module.

We also demand that the communication takes place at the module in which it was defined. This implies that *communicating atoms* cannot be exported. Due to this layered design in XP we do not need an alternative of the *consistency of communications* restriction as in PSF.

The encapsulation and abstraction operators are in practice strongly connected with the communication. From practical experience with PSF case studies we have learned that in a specification the encapsulation and abstraction operators are normally used only once per module, namely on the *top* process. On base of this knowledge we have tried in XP to let these operators coincide with module boundaries. This means that all processes in a module are put inside the scope of an encapsulation operator with an encapsulation set that consists of all communicating atoms from this module. Next an abstraction operator is applied with an abstraction set that consists of all atoms that are not exported.

We agree that this way of incorporating the encapsulation and abstraction operators in the modular structure might be too far-fetched. However, we firmly believe that the modular division of the communication function will lead to clearer specifications because communication possibilities can be decided by inspecting one module only.

9.3.4 IMPLEMENTATION

A syntax checker for XP has been implemented using the standard compiler construction tools Lex [LS79] and Yacc [Joh79]. Due to the careful design of the PSF Toolkit we have been able to reuse large parts of it in implementing XP.

The best way to make the toolkit available for XP would be to write a type checker and normalizer that produce TIL. However, the current implementation translates XP into M-TIL. This way we have been able to produce a prototype with less effort. This prototype is called *xp* and is available in the PSF Toolkit.

In implementing XP following this approach, we have also reused the PSF type checker and normalizer. However, there is one main drawback. Due to design goals of XP, *declare before use* and *full typing*, the type checker and normalizer for XP can be constructed more efficiently and can have better performance than the PSF equivalents. So by choosing for the implementation of XP through a rapid prototype, we lose efficiency.

9.4 A COMPARISON WITH OTHER TOOL SETS

There are several other tool sets available for different (algebraic) specification languages suited for specifying process behaviour. In this section we will give a comparison between some of those tools and the PSF Toolkit.

A very extensive survey of tools for Formal Description Techniques (FDT) and a good source for references is [CLV93]. Regretfully, the authors restrict themselves to publications pertaining to one of the three standardized languages: Estelle [ISO89b], LOTOS [ISO89a] and SDL [CCITT88].

9.4.1 TOOLS FOR LOTOS

The tools developed for LOTOS, especially within the LOTOSHERE project, have a great resemblance with the PSF Toolkit, although there is much wider variety of tools for LOTOS to choose from.

The Lite [vEij92] package from the LOTOSPHERE project supports besides a structure editor, a simulator and a program performing correctness preserving transformations, also a compiler tool, that translates (annotated) LOTOS into C and extensive report generators. The correctness preserving transformation tool can be compared with the PSF Proof Assistant.

Another interesting set of tools for LOTOS is LOEWE (LOTOS Engineering Workbench) developed at the IBM Zürich Research Laboratory [KBG93]. Some of the LOEWE tools are also part of the SPECS prototype environment. LOEWE offers tools for syntax and static semantic checking, simulation, state space exploration, code generation and model checking. The LOEWE tool set lays emphasis on generating executable code from formal specifications.

9.4.2 CONCURRENCY WORKBENCH

The Concurrency Workbench (CWB) [CPS89], developed at the University of Edinburgh, is a tool set supporting (T)CCS. The main differences with the PSF Toolkit are that the CWB supports only pure CCS, that is, there are no constructs supporting the use of data. The user interface is based on command lines in ASCII format rather than a windowing system.

Features present in the CWB but not in the PSF Toolkit are checking a specification against a proposition defined in a modal logic and deriving the missing part of an incomplete implementation given the specification that the implementation is to satisfy.

9.4.3 THE AUTO TOOL SET

At INRIA a set of tools has been implemented, which are based on automata (transition systems). This set consists of Ecrins [MdSV89], AUTO [Ver91] and Autograph [RdS91].

Ecrins is a system used in defining process algebras. It takes as input both the syntactical and semantical definition of the operators of a process algebra. The format used for defining the semantics of an operator, guarantees that the system is able to prove certain algebraic laws. If a process algebra has been defined using Ecrins, the system can be used to evaluate the behaviour of open terms and strong bisimulation laws.

AUTO is a verification tool based on reduction of finite automata networks using several types of bisimulations. Autograph is the graphical editor for automata. It can provide input for AUTO and it is able to display automata generated by AUTO.

The main difference between the INRIA tools and the PSF Toolkit is that the former are based on finite transition systems while the latter is based on recursive process definitions.

9.4.4 PROCESS ALGEBRA MANIPULATOR

The Process Algebra Manipulator (PAM) [Lin92] developed at the University of Sussex is quite similar to the Proof Assistant in the PSF Toolkit. Although PAM is a single tool and not a tool set, we discuss it here shortly. The main advantage of PAM over the PSF Proof Assistant is that PAM is able to operate on any set of axioms. This implies that many different process algebras can be manipulated with PAM, given that the algebra definition has been supplied.

9.5 PSF PAST, AN EVALUATION

In this section we will evaluate some of the design issues and targets of PSF. We will show where they were met and where not. Furthermore, we will explain the structure of the implementation project and discuss some implementation decisions.

9.5.1 DESIGN ISSUES

One of the main targets of the PSF project was to supply people working in the area of process algebras with a set of tools to support them in their specification activities. One of the initial results of PSF has been the integration of data, by means of algebraic specifications, into process specifications. Until that moment, data used in ACP specifications was treated in a rather ad hoc fashion. In order not to scare off people used to the syntax of ACP it was decided to develop a syntax for PSF that resembled the usual mathematical syntax. We succeeded in doing so, but in retrospect we must conclude that this cluttered up the language in some places (the use of sets for example) and lead to difficulties during the implementation phase. Unluckily, the people we aimed at initially turned out to be not the same as those who started using the tools.

Another important issue during the design of PSF has been to incorporate the *module* concept in process specifications. It was reasoned that in order to write specifications of reasonable size one should have the possibility to break up a specification into different modules. The solution to this was to borrow the modularization techniques used in ASF, partly because ASF was already integrated in PSF as the data specification language. Although this approach lead to some speed up in the initial phase of the project, it also has showed some shortcomings, partly because the modularization techniques available in ASF were not optimal and partly because they could not be transferred as easily to the process part.

9.5.2 PROJECT MANAGEMENT

In this section we will evaluate the project management regarding the implementation of the software for the PSF Toolkit as well as some implementation issues.

9.5.2.1 HARDWARE & SOFTWARE ISSUES

One of the important issues with respect to the implementation has been the choice of the hardware platform and the programming language to be used. For both issues compatibility and availability were important in order to have the possibility to cooperate and exchange software with other research teams and to be able to give software demonstrations at different sites.

The choice for SUN workstations as hardware platform was not difficult because this type of machine is in use at many research institutes. The UNIX operating system, used for this type of machine is used in many academic environments. This guarantees that porting the PSF Toolkit to other hardware platforms supporting UNIX is also possible.

The choice of the programming language gave some more problems. Experience with the Prolog implementation of the ASF system showed that execution speed was an issue. This made us decide to choose a traditional imperative programming language. Next there were several candidates. Because we were aware of the fact that we dealt with a project that was going to run for several years and would result in a large system, it was suggested that a programming language with good structuring mechanisms such as Ada or Modula could be beneficial.

We turned our attention to Ada for a while because it was reasoned that the *task* mechanism of Ada might be used to implement parallel PSF processes. Some experiments were carried out which showed that the granularity of the parallel processes was a deciding factor. The Ada implementation showed a satisfactory performance on a system with a few large processes, but lost an enormous amount of time to communication overhead when a fine-grained system with a large number of small processes was implemented. Because PSF specifications tend to be of the latter type Ada was found to be unsuitable to implement PSF processes by way of its *task* mechanism. Finally the availability and the cost of compilers for Ada and Modula made us decide to use plain old C, which is available on most UNIX systems, even though its structuring and modularization mechanisms are inferior to those of Modula and Ada.

9.5.2.2 Implementation Structure

The central interface for all tools in the PSF Toolkit is the Tool Interface Language TIL. Due to this design decision the different tools are stand-alone programs. This has as great advantage that the tools do not influence each other directly.

The different tools were implement by different (teams of) programmers in parallel. However, this structure has also disadvantages because reuse of software between programmers and different tools turned out to be low. Reuse has been achieved in standard routines for reading and writing TIL, in standard routines for rewriting and in some routines of the simulator and proof assistant. We experienced that the data structures used in one tool are not optimal or even not usable in the setting of another tool, so in this area we have seen less reuse.

Moreover, it turned out that at a later stage of the project the different tools *drifted* away from each other. It became more difficult for programmers to understand and change each other's code, because programmers were mainly working on their *own* program.

The problem that there was a small degree of interchangeability in software as described above, is also reflected in the technical documentation. Where this documentation is available, it mainly focuses on the algorithms involved and does not pay much attention to the data structures. We think that in this area a possible follow-up project could do better and we will indicate some possible solutions in a later section.

9.5.2.3 Graphical Interfaces

Computer programs can hardly do without a graphical user interface if they want to be taken seriously, nowadays. This places a heavy burden on the team implementing a tool, because the time spent to implement a good user interface can easily take up to 80% of the total time.

In the PSF Toolkit only two programs, the simulator and proof assistant, have a graphical user interface. The problem that we have encountered in this area is that the code necessary to implement the graphical user interface, tends to mingle with the code implementing the main functionality of a tool. This has as result that the tools are largely depending on one specific windowing system.

A different problem with the graphical user interface is that it currently is implemented using the X Window System, which is a relatively low-level set of graphical routines and user interface widgets. At the moment there are a number of alternative packages, such as OSF/Motif, offering higher level graphical user interface entities.

9.6 PSF Usage

In this section we will look at the projects and case studies in which PSF and the PSF Toolkit have been used.

9.6.1 Education

One of the first projects where the PSF Toolkit was used is in the Software Engineering course at the University of Amsterdam. Within this course the students were taught ASF as a formal specification language. Initially the Prolog-ASF implementation, described in chapter 2 of [Hen91], was used as a tool for the students to work out their practical

assignments. The performance of this ASF system was largely influenced by the Prolog implementation, which slowed down overall system performance because a rather large core image had to be loaded for each user running the system.

In 1989 the PSF Toolkit was introduced in the Software Engineering course. As described earlier, the data specification part of the PSF Toolkit can be used as an ASF implementation. The use of the PSF Toolkit significantly reduced compile and execution times. Moreover, the term rewriter *trs* was extended at a later stage with the option to scan equations in reverse order, which turned out to be of great benefit to the students. Using this option it is possible to detect unwanted order dependencies in an ASF specification.

Since 1991 the full functionality of the PSF Toolkit is used in education and its usage has been extended to the practical assignments of the *Process Algebra* course. Since 1993 the PSF Toolkit is also used in education at the Technical University of Eindhoven. Experience has shown that the performance of the tool kit is satisfactory, although a bottle-neck exists in the simulation of processes. This problem is largely due to the way the *sum* operator is handled within the tool kit.

9.6.2 CASE STUDIES

PSF and the PSF Toolkit have been used in several industrial case studies. A long-term case study is carried out in cooperation with DEC. The target in this project is the specification of a model (Computer Integrated Manufacturing) factory, which was built at DEC for educational and research purposes. One of the initial results was that through the PSF specification of the system a deadlock situation that occurred in the actual system could be located and removed. This case study started in 1992 and has resulted in the following four reports so far [VW92,HP92,PV93,BP93].

PSF and the PSF Toolkit have also been used to develop an algebraic specification of a traffic regulation system in cooperation with Nederland Haarlem B.V. [VW93] and to specify three practical protocols in the area of consumer electronics at Philips, Eindhoven.

Another area of interest for case studies has been the specification of communication protocols. Several of these specifications have been bundled into a book [MV93] which contains a series of PSF specifications of communication protocols at various levels of complexity, ranging from the Alternating Bit Protocol to the Token Ring Protocol.

9.6.3 RESEARCH

PSF, being the main computer implementation of the formalism ACP, has settled itself as one of the formalisms studied at the Programming Research Group (PRG) at the University of Amsterdam. The PSF Toolkit is used by a group of researchers developing PSF specifications for different case studies. This is illustrated by the fact that roughly a quarter of the technical reports that were issued by the PRG in the period 1987-1993 is dedicated to PSF, the PSF Toolkit and case studies in PSF. The PSF language is in use at several other Dutch universities as well and at the moment tool development is carried out in cooperation with the Technical University of Eindhoven.

9.7 PSF FUTURE

In this section we will try to look at the future of PSF and the PSF Toolkit. We will give some suggestions and recommendations for possible extensions of the current set of tools as well as possible follow-up languages and implementations.

9.7.1 TOOL EXTENSIONS

In this section we will discuss possible extensions to the PSF Toolkit. We will look at possibilities to speed up the processing of data, possible changes to the simulation process and the question of code generation from PSF specifications. Moreover we will discuss the position of graphical user interfaces and possible future representation techniques.

9.7.1.1 DATA TYPES

Experience has shown that term rewriting as implementation for data types in many cases turns into a bottle-neck with respect to execution speed. This is largely caused by the time needed for determining a rewrite rule that matches the term to be rewritten. Several approaches to overcome this bottle-neck are addressed in [Wal91].

The first technique is called *tree matching* and is related to the work described in [ODo85]. This technique is based on a compilational approach in which a set of rewrite rules is transformed into a tree matching automaton. Using this automaton it is possible to find a matching equation faster. In [Wal91] a typical speed-up factor *5* is mentioned for the total execution time of representative examples.

A second technique described in [Wal91], is the so-called *hybrid* implementation of algebraic specifications. In this case the rewriting framework is extended with functions that are implemented in a conventional programming language. This means that the terms that are operated on in such a system are of a hybrid nature: part of it is an algebraic term and part is represented as a data structure in a programming language. This technique offers the possibility to implement much used, low level routines such integer arithmetic and character string handling in an efficient way, whereas higher-level functions are still implemented by term rewriting. The implementation is *transparent*, which means that an external user is not aware of the fact that certain functions are not implemented by term rewriting. It is the responsibility of the implementor of such a function that the semantics complies with the rewrite rules for this function that must be given to act as interface between both implementations.

Apart from speed-up of the term rewriting procedure there are also possibilities to extend the term rewriter in the PSF Toolkit with the option to choose between different rewrite strategies (innermost/outermost) and facilities for (run time) cycle detection.

9.7.1.2 PROCESS SIMULATION

As addressed earlier, the current implementation of the simulator has emerged from an initially simple simulation algorithm, optimized towards a small number of process constructors. During the evolution of the PSF Toolkit many features were added to the initial implementation. Because most of these features needed special solutions, the total implementation has become rather complex.

To overcome some of the problems we experienced with this implementation, most notably modifiability and extendability, we suggest that a future simulator should be based on *one* clear concept. In our opinion term rewriting is a good candidate.

This seems a contradiction with respect to the bottle-neck that term rewriting can result in as we discussed in the previous section. However, term rewriting should not be implemented using a general term rewriting system, such as *trs*, but by a dedicated implementation. By this we mean that the rewrite rules are to be hard-coded, but still must recognizable as rewrite rules.

We experimented with this approach in the implementation of *trans* and our experience is that the program code is easier to understand because people can relate better to the general concept of term rewriting than the current specialized implementation. Moreover, we experienced that term rewriting offers a conceptual framework that makes it easier to add functionality in different stages.

9.7.1.3 TRANSITION SYSTEMS

In the previous section we mentioned the implementation of *trans*, which is strictly speaking not a simulator. However, generating transition systems from a specification and simulating a specification have a lot in common. We could argue that when we simulate a process, we only generate part of its transition system. This suggests that we could integrate simulation and transition system generation into one. To achieve this we see two possible solutions:

- Generate the complete transition system from a process specification and then simulate the transition system, this we will call the *compilational approach* as opposed to the currently used *interpretive approach*.

  The main advantage of the compilational approach is that simulating a transition system can be done much faster than simulating the process on a more abstract level as done in the interpretive approach. The reason for this is that a transition system already contains all possible execution paths in a very convenient format. Whereas, especially determining all possible communications slows down the simulation of processes in the interpretive approach. Another advantage is that it is possible to implement a special deadlock detection program that can indicate which states (processes) could possibly lead to a deadlock situation by following the traces leading to a deadlock state in backward direction.

  There are also some disadvantages to this approach. It can only be applied to finite processes, that is, processes generating a finite number of states. A second issue is the state space explosion. Transition systems tend to be very large, which means that internal memory (RAM) as well as background memory (hard disk) becomes a limiting factor. The final drawback is the longer time needed to generate the transition system, which relates to the traditional compiler versus interpreter trade-off.
- The second solution is a '*best of both worlds*' solution. This means that the transition system is generated '*on the fly*', that is, during the simulation. This implies that only that part of the transition system that is actually needed is generated. We will call this the *semi-interpretive* approach. To overcome the space problem as discussed for the compilational approach the number of stored states is bounded. If there is no more room to hold a newly generated state the *least recently* used state is removed to make space.

  The advantages of the semi-interpretive approach are clear: it can be applied to infinite processes too and space usage is limited to a controllable size.

Disadvantages are that only a sub-optimal execution speed can be achieved and that the deadlock detection possibilities are reduced to only part of the complete transition system.

Apart from the advantage in execution speed that transition systems have over an interpretive approach as discussed above, there is another more general advantage for transition systems, namely the possibility to exchange specifications between different tool sets. There are a number of tool sets based on different specification languages. This mostly implies that a possible interface between these tool sets must be created on the level of the process description language. On a lower level, however, these languages mostly rely on state transition systems. To open up interface possibilities between different tools a common format for transition systems (automata) has been proposed in [FMdS91].

As mentioned in section 6.5, the current implementation of *trans* only handles process specifications containing no data. For a real interface to other tools this restriction should be removed. A naive solution to this problem would be to create individual states for every possible combination of values for the data parameters of a state. This would mean that a process *S* with one boolean parameter would be turned into two different states namely: *S*[*true*] and *S*[*false*]. This approach would blow up the state space enormously and would restrict the implementation to finite data types only. Therefore we do not regard this to be a sufficient solution.

A more sophisticated approach would be to parameterize states with data types and to put conditions, evaluating the data parameters, on the transitions between states. More research has to be done into this area.

9.7.1.4 CODE GENERATION

One of the prerequisites for PSF and similar formalisms to ever attain industrial acceptance is that specifications ultimately can be compiled into, or at least linked with, traditional programming languages. There are several reasons for this requirement.

Although performance is mostly mentioned as the main criterion, there is another criterion at least as important namely the fact that one is only prepared to accept new technology if it builds upon and cooperates with existing applications. In this sense the PSF Toolkit and similar tools have a status comparable to the so-called CASE tools that have flooded the market in recent years. There are advocates who state that specification languages, and CASE tools for that matter, are only to be used in the design phase of a project and should not be used to generate code. We do not share this opinion because it inherently over-simplifies the process of software construction. If the software life-cycle could be divided into clear phases, for example: analysis-design-coding-testing, that are to be performed in a strict sequential fashion, the statement could hold. Unfortunately, however, in reality the software life-cycle consists of a constant re-iteration through one or more of these phases. This means that if there is no strict relation between the design and the implementation, design documents, just like general documentation, tend to get out of phase with the actual implementation.

In our opinion there is one way out of this problem, which is that one is forced to update one's design documents before changing the actual implementation. The crucial point now is, that if changing the design has no direct influence on the implementation updating the design is neglected. As a result, the design methodology and accompanying tools are not

considered helpful and are rejected. Therefore, it should be possible to generate, at least part of, the implementation from the design documents. In the setting of the PSF Toolkit we see two possible solutions to transform a specification into an implementation.

The first approach would be that a complete PSF specification is translated into a programming language. In this setting a PSF code generator would act as a kind of pre-processor in the same way as the language Eiffel [Mey88a,Mey88b] is implemented as a pre-processor generating C [KR88] code. For PSF a good candidate for the target language would be Occam [INMOS88] because Occam, being based on CSP [Hoa85], has many concepts in common with PSF. Although such a translation theoretically could yield a good performance and even actual concurrency, we see several problems.

It is not to be expected that the full functionality of PSF can be translated. This implies that a number of restrictions have to be kept in mind if code should be generated, thereby reducing the benefit of a specification language. Moreover, possible optimizations that have to be made on the level of the implementation language cannot be reflected in the PSF specification, which probably causes design and implementation to get out of phase.

The second, more promising, approach we see is to let conventional programming code be incorporated in the atomic actions of PSF. This *hybrid* approach is comparable with the solution chosen in the parser generator tool Yacc [Joh79] where the syntax definition rules can be interspersed with pieces of C code implementing user specified actions. In this solution the PSF system offers a framework for the high-level concurrent flow-of-control to which parts of lower-level code can be added in well-defined places. This approach also moves the problem of interaction with the operating system (input, output, file management &c.) from PSF to the conventional programming language where it has been solved already.

An example of this hybrid approach is the language *Scriptic* [vDel91] which initially supported a wide range of languages but currently focuses on combining C [KR88]/C++ [Str86] with elements that have been derived from process algebra to implement complex and concurrent control flow. The viewpoint in *Scriptic* is more from the programming language than from the (process) algebraic side. This implies that all data types are implemented using the local data types of the programming language at hand.

9.7.1.5 Graphical Interfaces

A second condition for tools to attain general acceptance nowadays, is that they are equipped with a graphical user interface. As addressed earlier, the code for the main functionality and the code for visualization are rather mingled in the current implementation of the PSF Toolkit. For any future implementation of the PSF Toolkit we strongly advise to strictly separate both portions of code. It might even be a good suggestion to define a special visualization interface that is to be called by the routines within the PSF Toolkit. At the other side of this interface these calls can then be translated into calls for different user interface packages. This would decouple the PSF Toolkit from one specific windowing system. Surprisingly, we experienced that among the different representation packages there is still a good place for a plain old ASCII menu driven interface, especially during tool development.

An ever recurring item on the wish list of users of the PSF Toolkit is a graphical simulation environment, extended with process animation. Although the current implementation of the simulator has a graphical interface, the actual simulation of process behaviour is only based on a trace of names of atomic actions. There is a clear need for a tool

that really visualizes the active atoms and processes. The AUTO Toolkit [Ver91] developed at INRIA contains such an interface and experiments have shown that such a visualization can be of great help in understanding complex process behaviour. It might be possible to incorporate this interface in the PSF Toolkit. The only drawback is that the AUTO tools operate on transition systems. So, experiments with integrating this tool could be started when *trans* is extended to incorporate data and its output is translated into the FC2 [FMdS91] format. This could be a temporary solution but we think that a proper simulation (animation) should be based on the higher level PSF specifications. To realize this a new tool, a graphical editor, should be developed in addition, allowing users to arrange the graphical images of their processes on the animation screen. Moreover, the simulator should be extended to highlight active entities on the animation screen. Regarding the expected effort to implement such a tool, we advise not to implement it on the basis of the current implementation of the simulator.

9.7.2 LANGUAGE ISSUES

9.7.2.1 SYNTAX CHANGES

In this section we will list some changes to the current version of PSF we would recommend for any follow-up languages. One of the clearest problems with PSF is the division between data and process modules. Although, conceptually it may seem a good idea to strictly separate the definition of data types and processes, in (PSF) practice it turns out to be a somewhat artificial division that forces the introduction of extra modules in a specification. These problems are described in detail in [Mul90].

- In many cases there is a clear need for process modules to use *auxiliary* functions on data types. Because these functions cannot be defined in the process module, a separate data module has to be defined solely for the purpose of providing these auxiliary functions. With the current modularization concepts there is an extra problem to this solution. The data module has to export the auxiliary functions for the process module. This implies that these functions automatically turn up in the export signature of the process module, thereby cluttering up the object name space. Especially, for *auxiliary* functions this is an unwanted feature.
- There is a similar problem with sets. Currently, sets can only be defined in the process section. This implies that if we want to introduce a data type and a set containing elements from this data type we have to define a special process module to construct the set. This construction has as side-effect that it is impossible to specify and export a set (of data) without exporting all its elements.

This strongly suggests that in a follow-up language the strict division between data and process modules should be removed, especially because this division has no further semantical implications. In our experiment with XP we did so and have encountered no specific problems.

A second suggestion that has no great implications is to change the notation for process operators from a binary infix to a prefix notation with no fixed number of arguments. The advantage is a clearer indentation structure, which especially pays off in case of complex process definitions for which, in the current version of PSF one easily loses track in counting

matching parentheses. The price to pay for the decision to use a prefix operator notation is a certain blow-up in vertical code size.

The final suggestion for a change in syntax is to group the equations defining a function. In doing so, we feel that a specification becomes clearer. In XP we even went a step further and require that a function is completely declared within one module, thereby avoiding that the meaning of a function can accidently be changed by additional equations in other modules.

A common critique to this solution is that one loses the ability to extend a function in an importing module. In our opinion this problem should be tackled in a way different from the current PSF solution. The object-oriented methodology offers a framework in which a sort and the functions operating on this sort can be changed into another similar sort through *inheritance* and subsequent *overriding* of functions. This newly created sort can then be extended without violating the above-mentioned principle of complete local definition.

9.7.2.2 LANGUAGE EXTENSIONS

The second category of changes to PSF would be to really extend the expressive power of the language. The foremost extension we see momentarily is the introduction of a notion of time.

In recent years many research articles have appeared that deal with this topic and we feel the time is right to evaluate the different suggestions and try to incorporate one of the proposals into PSF or its successors. The possibility to express time in a specification language is of course of utmost importance for the wide area of real-time applications and critical applications such as safety critical systems.

Another extension would be the incorporation of probability. This has also been a much-studied topic in previous years and we think that it would be right to experiment with probability in an implementation.

In [Mau91] a number of extensions to PSF are mentioned such as conditional choices, priorities and interrupts. Only the first suggestion has found its way into PSF until now in the form of *conditional expressions*. We think it would be useful to experiment with the other suggestions†.

The final suggestion for an extension is a more language-internal issue. In a number of PSF specifications the need for some kind of *state* variables and the possibility for a process to return data values to its *caller* became apparent. The initial suggestion to incorporate the state operator [BB88] was put aside in [Mul90] where it is shown that PSF as it is, has enough expressive power to specify a system that at first sight seems to call for the use of the state operator. The state of a system is recorded in an explicit process that runs in parallel with the main process. It is concluded in [Mul90] that the state operator as such seems to be a too general solution to the problem, but that the chaining operator [GV93] in combination with the *register operator* [Ver92a] seems to be an appropriate candidate for inclusion into PSF. In [Mau91] an actual proposal is given using so-called state variables. We think that some research still has to be done in this area, especially into the semantics of the different types of variables in PSF and whether it would be possible to unify them in some way. However, such an extension would be a valuable addition to PSF.

---

† Recently, PSF and the PSF Toolkit have been extended with *interrupts*, *disrupts* and *priorities* [Die94] as well as *iteration* and *nesting* [DP94].

9.7.2.3 MODULARIZATION TECHNIQUES

Finally we come to a much criticised area of PSF namely the modularization concepts. A number of these problems can already be identified in the concept as it is borrowed from ASF. However, there is a class of specific PSF problems that we encountered in developing case-studies for which we will give an example.

The *import* operator in ASF is used in two ways. The most common way is that a module *B* needs some sorts and functions from module *A* to use in its definitions. To this end module *B* can import module *A*. This is the most common view of import. However, if module *B* wants to extend the data types defined in module *A*, it also uses the *import* operator, possibly renaming the sorts and functions from module *A* to indicate that module *B* has a different view on these data types.

An example of this usage of import is when module *B* implements a *queue* data structure based on a *doubly-linked list* defined in module *A*. However, if module *B* should extend the imported data types in any way, this would also influence the semantics within module *A* and would result in a *non-persistent* specification. These problems automatically apply to the process part of PSF as well.

As addressed earlier, [Wie91] discusses the problem of *persistence* in algebraic specifications extensively. Determining whether a specification is persistent is non-trivial and to make sure that persistence can be checked by a computer implementation the language *Perspect* [Wie91] introduces a number of restrictions to an ASF-like language.

In the setting of the process part of PSF the problems described above can manifest themselves in yet another way, as a result of interference with the globally defined communication function. The communication function can be seen as an extra attribute of an atomic action and as such influences its semantics. A classical problem occurs in the definition of a buffer. Suppose we want to specify an *n*-element buffer as the concatenation of *n* one-element buffers. An intuitive solution is to specify the one-element buffer first, and then import *n* instances of this buffer into a module defining the communication between the individual buffers. Although we can rename the different instances of the one-element buffer there is in fact only one instance. If we would put the one-element buffers in parallel without defining a communication function, we would not notice this fact, and the system would behave as if it consisted of *n* individual buffers. However, as soon as we define a communication function for one of the (renamed) atomic actions it applies to all other atomic actions derived from the same origin as well. The compiler will issue an error in this situation because the communication between atomic actions is not defined in the module of origin. The following PSF specification is used to illustrate this problem.

```
process module Buf-1
begin

  exports
    begin
      atoms
        read, send
      processes
        Buf-1
    end

  definitions
     Buf-1 = read . send . Buf-1

end Buf-1
```

```
process module Buf-2
begin

  imports

    Buf-1 {
      renamed by [
        Buf-1 -> LeftBuf,
        read -> left-read,
        send -> left-send
      ]
    },

    Buf-1 {
      renamed by [
        Buf-1 -> RightBuf,
        read -> right-read,
        send -> right-send
      ]
    }

  atoms
    internal-communication

  processes
    Buf-2

  sets of atoms
    H = { internal-communication }

  communications
    left-send | right-read =
            internal-communication

  definitions
     Buf-2 = encaps(H,
              LeftBuf || RightBuf)

end Buf-2
```

The intended behaviour is illustrated in Figure 9.4. However, because there is only one instance of a *Buf-1* the specification actually describes the situation as depicted in Figure 9.5, which is clearly not the intended one.

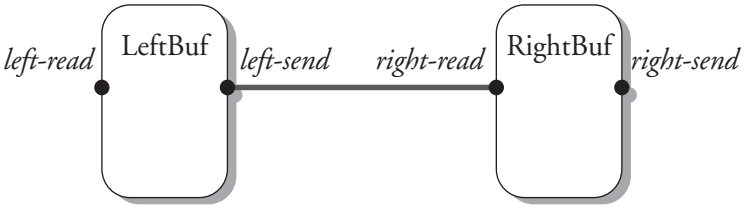


Figure 9.4 Intended communication scheme

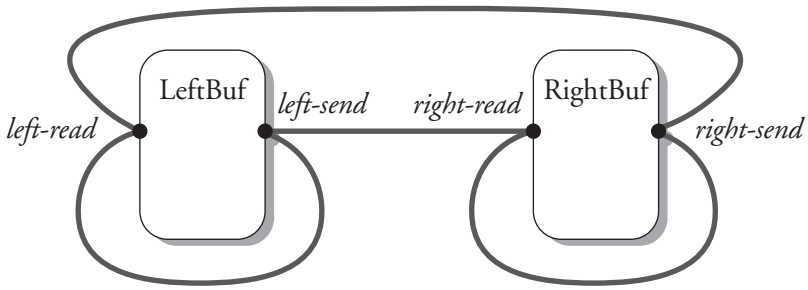


Figure 9.5 Actual communication scheme

To overcome the difficulties sketched above we suggest to strictly separate the two functions of the *import* operator. The second function of the import operator should be taken over by an *inheritance* mechanism, a well-known concept from the object-oriented methodology [Mey88a]. This would also enhance the overall expressiveness of the language.

9.7.3 NEW IMPLEMENTATIONS

In this section we will discuss the possibility of a new implementation of the PSF Toolkit. In our opinion it is the right time to stop adding new features to the PSF Toolkit as it is and start thinking of a redesign of the existing parts. We strongly believe that this design and implementation can benefit from the object-oriented methodology that gained more and more attention in recent years. Although this might seem a suggestion to jump the object-oriented bandwagon, we will try to give some reasons why it would be beneficial for the development of the PSF Toolkit.

In developing the PSF Toolkit we have gained a lot of knowledge in the field of specification languages and their implementations. On the basis of this detailed knowledge it must be possible to develop a more coherent design of the data model underlying the different tools, especially if we take into account that most tools manipulate the same relatively small set of objects. Based on experience we gained in other software projects we think that design methods such as OMT [Rum91] and Objectory [Jac92] offer a solid framework for such a redesign. These design methods start with a thorough description of the (data) objects in the system and the relations between different objects. This leads to a more stable design, which can hold for a longer period of time, because the objects in a system are less likely to change than the procedures operating on these objects. Expected benefits of a thorough object-oriented design are better design documentation and therefore better extendability and finally reusability of a larger part of the source code.

As a target language for a new implementation C++ [Str86] seems to be the perfect candidate. The language C++ has established itself in recent years and good compilers as well as extensive libraries with standard classes are available on most hardware platforms. Another advantage is that the transition from C to C++ can be performed gradually because C++ is largely backwards compatible with C. This avoids the need to disband the current implementation of the PSF Toolkit abruptly. We think that the disadvantages of an object-oriented implementation, loss in execution speed and the initially longer time needed to develop project specific classes and libraries will be outweighed by the expected benefits.

APPENDIX

APPENDIX A M-TIL SYNTAX

In this appendix we will give the definition of M-TIL in SDF (Syntax Definition Formalism) [HHKR89]. This 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 a combination of LEX [LS79] and YACC [Joh79].

```
module MTIL

  sorts
    Free-Format Free-Format-Char Comment-Char Digit Natural Id-First-Char
    Id-Char Id Tag Import-Tag Number Module-Id Parameter-Id Unknown-Id
    Renaming-Id Binding-Id Import-Id Sort-Id Function-Id Atom-Id
    Process-Id Set-Id Communication-Id Variable-Id Equation-Id
    Definition-Id Specification Module Module-Table Parameter-Table
    Unknown-Table Renaming-Table Binding-Table Import-Table Sort-Table
    Function-Table Atom-Table  Process-Table Set-Table
    Communication-Table Variable-Table  Equation-Table Definition-Table
    Module-Entry Parameter-Entry Unknown-Entry  Renaming-Entry
    Binding-Entry Import-Entry Sort-Entry Function-Entry  Atom-Entry
    Process-Entry Set-Entry Communication-Entry Variable-Entry
    Equation-Entry Definition-Entry Module-Index Parameter-Index
    Unknown-Index Renaming-Index Binding-Index Import-Index Sort-Index
    Function-Index  Atom-Index Process-Index Set-Index
    Communication-Index Variable-Index  Equation-Index Definition-Index
    ParObject-Index Variable-Type Set-Expr  Enumeration-Item Atom-Term
    Term Equation-Expr Definition-Expr Process-Head  Process-Expr Count

  lexical  syntax
    [ \n\t\f\r]                                  -> LAYOUT
    "--" Comment-Char* "\n"                      -> LAYOUT
    ~[\n]                                        -> Comment-Char
    ~[}]                                         -> Free-Format-Char
    "{" Free-Format-Char* "}"                    -> Free-Format
```

```
  [0-9]                                          -> Digit
  Digit+                                         -> Natural
  [a-zA-Z]                                       -> Id-First-Char
  [0-9a-zA-Z'\-]                                 -> Id-Char
  Id-First-Char Id-Char*                         -> Id
  [_!@?#$]                                       -> Tag
  [*BR]                                          -> Import-Tag
  "#"                                            -> Number
  "M"                                            -> Module-Id
  "P"                                            -> Parameter-Id
  "U"                                            -> Unknown-Id
  "R"                                            -> Renaming-Id
  "B"                                            -> Binding-Id
  "I"                                            -> Import-Id
  "1"                                            -> Sort-Id
  "2"                                            -> Function-Id
  "3"                                            -> Atom-Id
  "4"                                            -> Process-Id
  "5"                                            -> Set-Id
  "6"                                            -> Communication-Id
  "7"                                            -> Variable-Id
  "8"                                            -> Equation-Id
  "9"                                            -> Definition-Id

context-free  syntax

  Module+                                        -> Specification
  "module" Id Free-Format Module-Table Parameter-Table Unknown-Table
  Renaming-Table Binding-Table Import-Table Sort-Table Function-Table
  Atom-Table Process-Table Set-Table Communication-Table Variable-Table
  Equation-Table Definition-Table "end"          -> Module

  Number Natural                                 -> Count

  "table" Module-Id Count Module-Entry+ "end"    -> Module-Table
  "table" Parameter-Id Count Parameter-Entry+ "end"
                                                 -> Parameter-Table
  "table" Unknown-Id Count Unknown-Entry+ "end"
                                                 -> Unknown-Table
  "table" Renaming-Id Count Renaming-Entry+ "end"
                                                 -> Renaming-Table
  "table" Binding-Id Count Binding-Entry+ "end"
                                                 -> Binding-Table
  "table" Import-Id Count Import-Entry+ "end"    -> Import-Table
  "table" Sort-Id Count Sort-Entry+ "end"        -> Sort-Table
  "table" Function-Id Count Function-Entry+ "end"
                                                 -> Function-Table
  "table" Atom-Id Count Atom-Entry+ "end"        -> Atom-Table
  "table" Process-Id Count Process-Entry+ "end"
                                                 -> Process-Table
  "table" Set-Id Count Set-Entry+ "end"          -> Set-Table
  "table" Communication-Id Count Communication-Entry+ "end"
                                                 -> Communication-Table
  "table" Variable-Id Count Variable-Entry+ "end"
                                                 -> Variable-Table
```

```
"table" Equation-Id Count Equation-Entry+ "end"
                                          -> Equation-Table
"table" Definition-Id Count Definition-Entry+ "end"
                                          -> Definition-Table

                                          -> Module-Table
                                          -> Parameter-Table
                                          -> Unknown-Table
                                          -> Renaming-Table
                                          -> Binding-Table
                                          -> Import-Table
                                          -> Sort-Table
                                          -> Function-Table
                                          -> Atom-Table
                                          -> Process-Table
                                          -> Set-Table
                                          -> Communication-Table
                                          -> Variable-Table
                                          -> Equation-Table
                                          -> Definition-Table

Module-Index Free-Format                  -> Module-Entry
Parameter-Index Tag Natural ParObject-Index* Free-Format
                                          -> Parameter-Entry
Unknown-Index Free-Format                 -> Unknown-Entry
Renaming-Index Unknown-Index Unknown-Index Free-Format
                                          -> Renaming-Entry
Binding-Index Parameter-Index Module-Index Natural Renaming-Index+
Free-Format                               -> Binding-Entry
Import-Index Module-Index Import-Tag Natural Binding-Index* Natural
Renaming-Index* Free-Format               -> Import-Entry
Sort-Index Tag Free-Format                -> Sort-Entry
Function-Index Tag Natural Sort-Index* Natural Sort-Index*
Free-Format                               -> Function-Entry
Atom-Index Tag Natural Sort-Index* Free-Format
                                          -> Atom-Entry
Process-Index Tag Natural Sort-Index* Free-Format
                                          -> Process-Entry
Set-Index Tag Sort-Index Set-Expr Free-Format
                                          -> Set-Entry
Communication-Index
Atom-Term Atom-Term Atom-Term Free-Format  -> Communication-Entry
Variable-Index Tag Variable-Type Free-Format  -> Variable-Entry
Equation-Index Equation-Expr Free-Format  -> Equation-Entry
Definition-Index Definition-Expr Free-Format  -> Definition-Entry

"[M." Natural "]"                         -> Module-Index
"[P." Natural "]"                         -> Parameter-Index
"[U." Natural "]"                         -> Unknown-Index
"[R." Natural "]"                         -> Renaming-Index
"[B." Natural "]"                         -> Binding-Index
"[I." Natural "]"                         -> Import-Index
"[1." Natural "]"                         -> Sort-Index
"[2." Natural "]"                         -> Function-Index
"[3." Natural "]"                         -> Atom-Index
```

```
"[4." Natural "]"                                -> Process-Index
"[5." Natural "]"                                -> Set-Index
"[6." Natural "]"                                -> Communication-Index
"[7." Natural "]"                                -> Variable-Index
"[8." Natural "]"                                -> Equation-Index
"[9." Natural "]"                                -> Definition-Index

"[1.?]"                                          -> Sort-Index
"[2.?]"                                          -> Function-Index
"[3.?]"                                          -> Atom-Index
"[4.?]"                                          -> Process-Index
"[5.?]"                                          -> Set-Index

Sort-Index                                       -> ParObject-Index
Function-Index                                   -> ParObject-Index
Atom-Index                                       -> ParObject-Index
Process-Index                                    -> ParObject-Index
Set-Index                                        -> ParObject-Index

Sort-Index                                       -> Variable-Type
Set-Index                                        -> Variable-Type

Set-Index                                        -> Set-Expr
"<:," Natural ">(" Enumeration-Item+ ")"         -> Set-Expr
"<+," Natural ">(" Set-Expr+ ")"                 -> Set-Expr
"<.," Natural ">(" Set-Expr+ ")"                 -> Set-Expr
"<\," Natural ">(" Set-Expr+ ")"                 -> Set-Expr

Term                                             -> Enumeration-Item
Atom-Term                                        -> Enumeration-Item

Atom-Index                                       -> Atom-Term
Atom-Index "(" Term+ ")"                         -> Atom-Term

Variable-Index                                   -> Term
Function-Index                                   -> Term
Function-Index "(" Term+ ")"                     -> Term

Term "=" Term                                    -> Equation-Expr

Process-Head "=" Process-Expr                    -> Definition-Expr

Process-Index                                    -> Process-Head
Process-Index "(" Term+ ")"                      -> Process-Head

Process-Head                                     -> Process-Expr
Atom-Term                                        -> Process-Expr
"<i>"                                            -> Process-Expr
"<e>(" Set-Index Process-Expr ")"                -> Process-Expr
"<h>(" Set-Index Process-Expr ")"                -> Process-Expr
"<A>(" Variable-Index Process-Expr ")"           -> Process-Expr
"<P>(" Variable-Index Process-Expr ")"           -> Process-Expr
"<a," Natural ">(" Process-Expr+ ")"             -> Process-Expr
"<s," Natural ">(" Process-Expr+ ")"             -> Process-Expr
"<p," Natural ">(" Process-Expr+ ")"             -> Process-Expr
"<f>(" Term Term Process-Expr ")"                -> Process-Expr
```

APPENDIX B I-TIL SYNTAX

In this section we will give the definition of the I-TIL syntax in SDF.

```
module ITIL

  sorts
    Free-Format Free-Format-Char Comment-Char Digit Natural Id-First-Char
    Id-Char Id Tag Number Module-Id Parameter-Id Unknown-Id Renaming-Id
    Binding-Id Import-Id Sort-Id Function-Id Atom-Id Process-Id Set-Id
    Communication-Id Variable-Id Equation-Id Definition-Id

    Specification Module Module-Table Parameter-Table Unknown-Table
    Renaming-Table Binding-Table Import-Table Sort-Table Function-Table
    Atom-Table Process-Table Set-Table Communication-Table Variable-Table
    Equation-Table Definition-Table Module-Entry Parameter-Entry
    Unknown-Entry Renaming-Entry Binding-Entry Import-Entry Sort-Entry
    Function-Entry Atom-Entry Process-Entry Set-Entry Communication-Entry
    Variable-Entry Equation-Entry Definition-Entry Module-Index
    Parameter-Index Unknown-Index Renaming-Index Binding-Index
    Import-Index Sort-Index Function-Index Atom-Index Process-Index
    Set-Index Communication-Index Variable-Index Equation-Index
    Definition-Index ParObject-Index Variable-Type Set-Expr
    Enumeration-Item Atom-Term Term Equation-Expr Definition-Expr
    Process-Head Process-Expr Count

  lexical syntax
    [ \n\t\f\r]                                  -> LAYOUT
    "--" Comment-Char* "\n"                      -> LAYOUT
    ~[\n]                                        -> Comment-Char
    ~[}]                                         -> Free-Format-Char
    "{" Free-Format-Char* "}"                    -> Free-Format
    [0-9]                                        -> Digit
    Digit+                                       -> Natural
    [a-zA-Z]                                     -> Id-First-Char
    [0-9a-zA-Z'\-]                               -> Id-Char
    Id-First-Char Id-Char*                       -> Id
    [_!@?#$]                                     -> Tag
    "#"                                          -> Number
    "M"                                          -> Module-Id
    "P"                                          -> Parameter-Id
    "1"                                          -> Sort-Id
    "2"                                          -> Function-Id
    "3"                                          -> Atom-Id
    "4"                                          -> Process-Id
    "5"                                          -> Set-Id
    "6"                                          -> Communication-Id
    "7"                                          -> Variable-Id
    "8"                                          -> Equation-Id
    "9"                                          -> Definition-Id
```

```
context-free  syntax

  Module+                                          -> Specification
  "module" Id Free-Format Module-Table Parameter-Table Sort-Table
  Function-Table Atom-Table Process-Table Set-Table
  Communication-Table Variable-Table Equation-Table Definition-Table
  "end"                                            -> Module

  Number Natural                                   -> Count

  "table" Module-Id Count Module-Entry+ "end"      -> Module-Table
  "table" Parameter-Id Count Parameter-Entry+ "end"
                                                   -> Parameter-Table
  "table" Sort-Id Count Sort-Entry+ "end"          -> Sort-Table
  "table" Function-Id Count Function-Entry+ "end"
                                                   -> Function-Table
  "table" Atom-Id Count Atom-Entry+ "end"          -> Atom-Table
  "table" Process-Id Count Process-Entry+ "end"
                                                   -> Process-Table
  "table" Set-Id Count Set-Entry+ "end"            -> Set-Table
  "table" Communication-Id Count Communication-Entry+ "end"
                                                   -> Communication-Table
  "table" Variable-Id Count Variable-Entry+ "end"
                                                   -> Variable-Table
  "table" Equation-Id Count Equation-Entry+ "end"
                                                   -> Equation-Table
  "table" Definition-Id Count Definition-Entry+ "end"
                                                   -> Definition-Table

                                                   -> Module-Table
                                                   -> Parameter-Table
                                                   -> Sort-Table
                                                   -> Function-Table
                                                   -> Atom-Table
                                                   -> Process-Table
                                                   -> Set-Table
                                                   -> Communication-Table
                                                   -> Variable-Table
                                                   -> Equation-Table
                                                   -> Definition-Table

  Module-Index Free-Format                           -> Module-Entry
  Parameter-Index Tag Natural ParObject-Index* Free-Format
                                                   -> Parameter-Entry
  Sort-Index Tag Free-Format                       -> Sort-Entry
  Function-Index Tag Natural Sort-Index* Natural Sort-Index*
  Free-Format                                      -> Function-Entry
  Atom-Index Tag Natural Sort-Index* Free-Format
                                                   -> Atom-Entry
  Process-Index Tag Natural Sort-Index* Free-Format
                                                   -> Process-Entry
  Set-Index Tag Sort-Index Set-Expr Free-Format
                                                   -> Set-Entry
  Communication-Index
  Atom-Term Atom-Term Atom-Term Free-Format         -> Communication-Entry
  Variable-Index Tag Variable-Type Free-Format     -> Variable-Entry
  Equation-Index Equation-Expr Free-Format         -> Equation-Entry
  Definition-Index Definition-Expr Free-Format     -> Definition-Entry
```

```
"[" Natural ".M." Natural "]"                -> Module-Index
"[" Natural ".P." Natural "]"                -> Parameter-Index
"[" Natural ".1." Natural "]"                -> Sort-Index
"[" Natural ".2." Natural "]"                -> Function-Index
"[" Natural ".3." Natural "]"                -> Atom-Index
"[" Natural ".4." Natural "]"                -> Process-Index
"[" Natural ".5." Natural "]"                -> Set-Index
"[" Natural ".6." Natural "]"                -> Communication-Index
"[" Natural ".7." Natural "]"                -> Variable-Index
"[" Natural ".8." Natural "]"                -> Equation-Index
"[" Natural ".9." Natural "]"                -> Definition-Index

Sort-Index                                   -> ParObject-Index
Function-Index                               -> ParObject-Index
Atom-Index                                   -> ParObject-Index
Process-Index                                -> ParObject-Index
Set-Index                                    -> ParObject-Index

Sort-Index                                   -> Variable-Type
Set-Index                                    -> Variable-Type


                                             -> Set-Expr
Set-Index                                    -> Set-Expr
"<:," Natural ">(" Enumeration-Item+ ")"     -> Set-Expr
"<+," Natural ">(" Set-Expr+ ")"             -> Set-Expr
"<.," Natural ">(" Set-Expr+ ")"             -> Set-Expr
"<\," Natural ">(" Set-Expr+ ")"             -> Set-Expr

Term                                         -> Enumeration-Item
Atom-Term                                    -> Enumeration-Item

Atom-Index                                   -> Atom-Term
Atom-Index "(" Term+ ")"                     -> Atom-Term

Variable-Index                               -> Term
Function-Index                               -> Term
Function-Index "(" Term+ ")"                 -> Term

Term "=" Term                                -> Equation-Expr

Process-Head "=" Process-Expr                -> Definition-Expr

Process-Index                                -> Process-Head
Process-Index "(" Term+ ")"                  -> Process-Head

Process-Head                                 -> Process-Expr
Atom-Term                                    -> Process-Expr
"<i>"                                        -> Process-Expr
"<e>(" Set-Index Process-Expr ")"            -> Process-Expr
"<h>(" Set-Index Process-Expr ")"            -> Process-Expr
"<A>(" Variable-Index Process-Expr ")"       -> Process-Expr
"<P>(" Variable-Index Process-Expr ")"       -> Process-Expr
"<a," Natural ">(" Process-Expr+ ")"         -> Process-Expr
"<s," Natural ">(" Process-Expr+ ")"         -> Process-Expr
"<p," Natural ">(" Process-Expr+ ")"         -> Process-Expr
"<f>(" Term Term Process-Expr ")"            -> Process-Expr
```

APPENDIX C XP SYNTAX

In this section we give a syntax definition of the XP language based on the *Lex* and *Yacc* grammars of the actual implementation. The bold terminal symbols in the *Yacc* grammar are literals whereas the italic terminal symbols are tokens from the *Lex* grammar.

```
/* Lex grammar */

ident               [A-Za-z0-9\_\-]+
newline             [\r\f\n]
layout              [ \t\r\f\n]
comment             [@][^\n]*[\n]

%%

{ident}             { return(id); }
{ident}/"("         { return(func_id); }
{ident}/"<"         { return(atom_id); }
{ident}/"["         { return(proc_id); }
{layout}{comment}   ;
^{comment}          ;
[ \t]               ;


/* Yacc grammar */

module
  : module
    id
    signature
    end
  ;

signature
  : introduction
  | signature introduction
  ;

introduction
  : sort_intro
  | func_intro
  | atom_intro
  | proc_intro
  ;


/* rules for sorts */

sort_intro
  : sort sort_entries
  ;
```

```
sort_entries
  : sort_entry
  | sort_entries sort_entry
  ;

sort_entry
  : sort_decl sort_def visibility ;
  ;

sort_decl
  : id
  ;

sort_def
  : { import }
  | { }
  | /* empty */
  ;

/* rules for functions */

func_intro
  : func func_entries
  ;

func_entries
  : func_entry
  | func_entries func_entry
  ;

func_entry
  : func_decl func_def visibility ;
  ;

func_decl
  : func_id ( input_type ) : output_type
  ;

func_def
  : { variables equ_defs }
  | { import }
  | { }
  | /* empty */
  ;


/* rules for atoms */

atom_intro
  : atom atom_entries
  ;
```

```
atom_entries
   : atom_entry
   | atom_entries atom_entry
   ;

atom_entry
   : atom_decl atom_def visibility ;
   ;

atom_decl
   : atom_id < input_type >
   ;

atom_def
   : { variables comm_defs }
   | { import }
   | { }
   | /* empty */
   ;

/* rules for processes */

proc_intro
   : proc proc_entries
   ;

proc_entries
   : proc_entry
   | proc_entries proc_entry
   ;

proc_entry
   : proc_decl proc_def visibility ;
   ;

proc_decl
   : proc_id [ input_type ]
   ;

proc_def
   : { variables process_def }
   | { import }
   | { }
   | /* empty */
   ;

/* rules for modularity constructs */

visibility
   : ->
   | -|
   | /* empty */
   ;
```

```
import
  : <- import_object
  ;

import_object
  : id
  | id . id
  ;


/* rules for variables */

variables
  : for var_defs
  | /* empty */
  ;

var_defs
  : var_def
  | var_defs var_def
  ;

var_def
  : id : id ;
  ;


/* rules for types */

input_type
  : id
  | input_type , id
  | /* empty */
  ;

output_type
  : id
  ;

/* rules for equations */

equ_defs
  : equ_def
  | equ_defs equ_def
  ;

equ_def
  : equation ;
  | cond_equation ;
  ;

equation
  : lhs_eq = rhs_eq
  ;
```

```
cond_equation
  : equation when equation_list
  ;


equation_list
  : equation
  | equation_list , equation
  ;

lhs_eq
  : func_id ( data_term_list )
  ;

rhs_eq
  : data_term
  ;

data_term
  : id
  | func_id ( data_term_list )
  ;

data_term_list
  : data_term
  | data_term_list , data_term
  | /* empty */
  ;


/* rules for communication definition */

comm_defs
  : comm_def
  | comm_defs comm_def
  ;

comm_def
  : comm_atom = comm_atom | comm_atom ;
  ;

comm_atom
  : atom_id < par_list >
  ;

par_list
  : id
  | par_list , id
  | /* empty */
  ;
```

```
/* rules for process definitons */

process_def
   : process_head = guarded_process_exp
   ;

process_head
   : proc_id [ par_list ]
   ;

process_term
   : guarded_process_exp
   | process_term guarded_process_exp
   ;

guarded_process_exp
   : process_exp
   | guard -> process_exp
   ;

process_exp
   : atom_id < data_term_list > ;
   | proc_id [ data_term_list ] ;
   | skip ;
   | asp_operator { variables process_term }
   ;

asp_operator
   : alt
   | seq
   | par
   ;

guard
   : [ equation ]
   ;
```

APPENDIX D AN XP EXAMPLE

In this appendix we will give an impression of the usage of XP. To this end we present a specification of an alternating bit protocol.

```
module Booleans

  sort
    Boolean {};

  func
    true(): Boolean {};
    false(): Boolean {};

    not(Boolean): Boolean
    {
      not(true()) = false();
      not(false()) = true();
    };

    and(Boolean,Boolean): Boolean
    {
    for x: Boolean;
      and(true(),x) = x;
      and(false(),x) = false();
    };

    or(Boolean,Boolean): Boolean
    {
    for x: Boolean;
      or(true(),x) = true();
      or(false(),x) = x;
    };

end


module Data

  sort
    Data;

  func
    0() : Data;
    1() : Data;
    data_error() : Data;

end
```

```
module Frames

  sort
    Frame;
    Boolean  { <- Booleans };
    Data     { <- Data };

  func
    frame(Boolean,Data) : Frame;
    frame_error() : Frame;

end


module Acknowledgements

  sort
    Boolean  { <- Booleans };
    Ack;

  func
    ack(Boolean) : Ack;
    ack_error() : Ack;

end


module Sender

  sort
    Boolean  { <- Booleans };
    Data     { <- Data };
    Frame    { <- Frames };
    Ack      { <- Acknowledgements };

  func
    false() : Boolean            { <- Booleans };
    not(Boolean) : Boolean       { <- Booleans };
    frame(Boolean,Data) : Frame  { <- Frames };
    ack_error() : Ack            { <- Acknowledgements };
    ack(Boolean) : Ack           { <- Acknowledgements };

  atom
    receive_data<Data> {} ->;
    send_frame<Frame> {} ->;
    receive_ack<Ack> {} ->;

  proc
    RM[Boolean];
    S[] {
    S[] =
      RM[false()];
    } ->;
```

```
    SF[Boolean,Data];
    RM[Boolean] {
    for b:Boolean;
    RM[b] =
      alt {
      for d:Data;
        seq {
          receive_data<d>;
          SF[b,d];
        }
      }
    };

    RA[Boolean,Data];
    SF[Boolean,Data] {
    for b:Boolean; d:Data;
    SF[b,d] =
      seq {
        send_frame<frame(b,d)>;
        RA[b,d];
      }
    };

    RA[Boolean,Data] {
    for b:Boolean; d:Data;
    RA[b,d] =
      alt {
        seq {
          alt {
            receive_ack<ack_error()>;
            receive_ack<ack(not(b))>;
          }
          SF[b,d];
        }
        seq {
          receive_ack<ack(b)>;
          RM[not(b)];
        }
      }
    };

end


module Frame_Channel

  sort
    Frame  { <- Frames };

  func
    frame_error() : Frame  { <- Frames };
```

```
  atom
    receive_frame<Frame> {} ->;
    send_frame<Frame> {} ->;

  proc
    K[]
    {
    K[] =
      seq {
        alt {
        for f:Frame;
          receive_frame<f>;
          alt {
            seq {
              skip;
              send_frame<f>;
            }
            seq {
              skip;
              send_frame<frame_error()>;
            }
          }
          K[];
        }
      }
    } ->;

end


module Receiver

  sort
    Boolean  { <- Booleans };
    Data     { <- Data };
    Frame    { <- Frames };
    Ack      { <- Acknowledgements };

  func
    false() : Boolean            { <- Booleans };
    not(Boolean) : Boolean       { <- Booleans };
    frame_error() : Frame        { <- Frames };
    frame(Boolean,Data) : Frame  { <- Frames };
    ack(Boolean) : Ack           { <- Acknowledgements };

  atom
    receive_frame<Frame> ->;
    send_data<Data> ->;
    send_ack<Ack> ->;
```

```
proc

  RF[Boolean];
  R[] {
  R[] =
    RF[false()];
  } ->;

  SA[Boolean];
  SM[Boolean,Data];
  RF[Boolean] {
  for b: Boolean;
  RF[b] =
    alt {
      seq {
        alt {
        for d:Data;
          receive_frame<frame(not(b),d)>;
          receive_frame<frame_error()>;
        }
        SA[not(b)];
      }
      alt {
      for d:Data;
        seq {
          receive_frame<frame(b,d)>;
          SM[b,d];
        }
      }
    }
  };

  SA[Boolean] {
  for b : Boolean;
  SA[b] =
    seq {
      send_ack<ack(b)>;
      RF[not(b)];
    }
  };

  SM[Boolean,Data] {
  for b: Boolean; d: Data;
  SM[b,d] =
    seq {
      send_data<d>;
      SA[b];
    }
  };

end
```

```
module Ack_Channel

  sort
    Ack {<- Acknowledgements};

  func
    ack_error() : Ack {<- Acknowledgements};

  atom
    receive_ack<Ack> {} ->;
    send_ack<Ack> {} ->;

  proc
    L[]
    {
      L[] =
      seq {
        alt {
        for a:Ack;
          receive_ack<a>;
          alt {
            seq {
              skip;
              send_ack<a>;
            }
            seq {
              skip;
              send_ack<ack_error()>;
            }
          }
        }
        L[];
      }
    } ->;

end


module ABP

  sort
    Data   { <- Data };
    Frame  { <- Frames };
    Ack    { <- Acknowledgements };

  atom
    input<Data>            { <- Sender.receive_data };
    output<Data>           { <- Receiver.send_data };
    frame_to_ch<Frame>     { <- Sender.send_frame };
    frame_from_snd<Frame>  { <- Frame_Channel.receive_frame };
    frame_to_rec<Frame>    { <- Frame_Channel.send_frame };
    frame_from_ch<Frame>   { <- Receiver.receive_frame };
    ack_to_ch<Ack>         { <- Receiver.send_ack };
    ack_from_rec<Ack>      { <- Ack_Channel.receive_ack };
```

```
    ack_to_snd<Ack>        { <- Ack_Channel.send_ack };
    ack_from_ch<Ack>       { <- Sender.receive_ack };

    frame_in_ch<Frame> {
      for f:Frame;
      frame_in_ch<f> = frame_to_ch<f> | frame_from_snd<f>;
    };

    frame_in_rec<Frame> {
      for f:Frame;
      frame_in_rec<f> = frame_to_rec<f> | frame_from_ch<f>;
    };

    ack_in_ch<Ack> {
      for a:Ack;
      ack_in_ch<a> = ack_to_ch<a> | ack_from_rec<a>;
    };

    ack_in_snd<Ack> {
      for a:Ack;
      ack_in_snd<a> = ack_to_snd<a> | ack_from_ch<a>;
    };

  proc
    S[]   { <- Sender };
    K[]   { <- Frame_Channel };
    R[]   { <- Receiver };
    L[]   { <- Ack_Channel };

    ABP[] {
    ABP[] =
      par {
        S[]; K[]; R[]; L[];
      }
    };


end
```

NEDERLANDSE SAMENVATTING

Sinds de ontwikkeling van de eerste computer hebben de formalismen die gebruikt worden om computers te programmeren zich tot een steeds abstracter beschrijvingsniveau ontwikkeld. De eerste programmeurs moesten nog de bit-representatie van de intern gebruikte machinecode kennen en programmeerden een computer uitsluitend door middel van rijtjes enen en nullen. Het was duidelijk, dat dit niet de meest optimale manier van programmeren was en al snel werden zogenaamde *assembler*-talen ontwikkeld als het eerste niveau van abstractie.

In een assembler-taal worden machine-instructies gerepresenteerd door afgekorte namen (*mnemonics*). Om deze representatie te vertalen in machinecode wordt gebruik gemaakt van een hulpprogramma, de *assembler*. Een assembler kan daarom beschouwd worden als het eerste computerhulpmiddel (*tool*) voor het ontwikkelen van computerprogramma's.

Assembler-talen zijn heden ten dage nog steeds in gebruik in speciale toepassingsgebieden. Een groot nadeel van assembler-talen is, naast het lage abstractieniveau, dat elke processor(familie) met zijn eigen specifieke assembler-taal geprogrammeerd moet worden, daar deze taal direct op de architectuur van de processor gebaseerd is.

Het volgende abstractieniveau werd bereikt door de introductie van de eerste machine-onafhankelijke programmeertalen zoals *Fortran* en *LISP* aan het einde van de jaren vijftig en het begin van de jaren zestig. Tegelijk met deze talen werden twee typen tools geïntroduceerd om programma's geschreven in deze hogere programmeertalen te verwerken. Het eerste type hulpmiddel is een *compiler*, die een compleet programma vertaalt in executeerbare machinecode. Een andere benadering wordt gevolgd door een *interpreter* die slechts één instructie (*statement*) uit de hogere programmertaal tegelijkertijd verwerkt. Nadat een interpreter een statement heeft geanalyseerd, worden de bijbehorende machine-instructies gegenereerd, deze machine-instructies worden door de processor uitgevoerd en vervolgens wordt het volgende *statement* opgehaald.

De ontwikkeling van programmeertalen heeft zich voortgezet tot op de dag van vandaag. Dit heeft geleid tot steeds verder geperfectioneerde programmeertalen, waaronder vele voor specifieke toepassingsgebieden, en uiteindelijk tot moderne algemeen toepasbare talen zoals *Ada* en *C++*.

Hoewel programmeertalen steeds krachtiger werden en een groeiend aantal structureringsmechanismen konden aanbieden, groeide reeds in het begin van de jaren

zeventig het besef, dat men door programmeren alleen waarschijnlijk niet in staat zou zijn, de steeds ingewikkelder wordende software-systemen met succes te kunnen implementeren. Tegen deze achtergrond onstond een stroming die er voor pleitte om eerst een specificatie van een probleem te geven, alvorens men tot de implementatie overgaat. De eerste specificatietalen die als gevolg hiervan onstonden, waren sterk op de syntax van programmeertalen zowel als op het onderliggende executiemodel geörienteerd. Later werden ook specificatietalen ontwikkeld, die op andere executiemodellen gebaseerd waren zoals *termherschrijfsystemen* of zelfs op *propositionele logica* of *eerste-orde logica*. Hierdoor werden beschrijvingen op een nog abstracter niveau en met mathematische precisie mogelijk.

De resultaten die in dit proefschrift gepresenteerd worden, behoren duidelijk tot deze categorie specificatietalen, maar breiden deze klasse echter uit met één nieuw concept: *concurrency* (parallellisme). De eerder gegeven beschrijvingen gingen uit van programmeertalen die gebaseerd zijn op computers met één enkele centrale processor. Met de komst van parallelle computers, waarin verschillende processoren tegelijkertijd samenwerken, werden naast nieuwe programmeerconstructies ook nieuwe problemen geïntroduceerd. Daar parallellisme een centraal thema vormt in dit proefschrift zullen we de aard van deze problematiek in het onderstaande kort toelichten.

Een groot deel van de complexiteit die onstaat bij de verwerking van parallelle processen, komt voort uit het feit dat zelfs een eenvoudig parallel systeem zich in een groot aantal verschillende toestanden kan bevinden. We zullen dit verduidelijken aan de hand van een voorbeeld. We stellen ons voor dat we een kruispunt met vier stoplichten moeten beschrijven. Elk stoplicht kan zich in één van de volgende toestanden bevinden: *rood*, *oranje* of *groen*. Als we de combinatie van vier stoplichten die tegelijkertijd actief zijn beschouwen, kan het totale systeem zich theoretisch in één van de 3^4 $(3 \times 3 \times 3 \times 3) = 81$ mogelijke toestanden bevinden. Dit aantal van *81* toestanden wordt alleen bereikt als de stoplichten volledig onafhankelijk van elkaar functioneren. In de praktijk zouden veel van deze toestanden echter tot verkeersopstoppingen of erger nog tot ongelukken leiden. In een realistische specificatie van een kruispunt beogen we daarom een veel kleiner aantal *veilige* toestanden.

Als we de toestanden van het kruispunt coderen door middel van de toestanden van de stoplichten (r, o, g) langs de noordelijke, oostelijke, zuidelijke en westelijke tak van het kruispunt, in precies die volgorde, vinden we de volgende vijf veilige toestanden: $\langle g,r,g,r \rangle$, $\langle o,r,o,r \rangle$, $\langle r,r,r,r \rangle$, $\langle r,g,r,g \rangle$ en $\langle r,o,r,o \rangle$. Uit het voorbeeld moge blijken dat er behoefte bestaat aan een systematische methode, die in staat is om dergelijke parallelle processen te beschrijven en die de mogelijkheid biedt bepaalde eigenschappen van een systeem te verifiëren.

Gedurende de laatste twee decennia zijn er verschillende voorstellen gedaan voor formalismen die in staat zijn het gedrag van parallelle systemen te beschrijven. Het materiaal in dit proefschrift is gebaseerd op één van deze formalismen genaamd: *Algebra of Communicating Processes* (ACP). De ontwikkeling van ACP begon in 1982 aan het *Centrum voor Wiskunde en Informatica* (CWI) te Amsterdam. Vijf jaar later, in 1987, werd de basis gelegd voor de implementatie van een verzameling tools door het ontwerp van de specificatietaal PSF (*Process Specification Formalism*). Dit proefschrift beschrijft de *PSF*

Toolkit, een verzameling computerhulpmiddelen, die verder onderzoek op dit gebied heeft voortgebracht. Het proefschrift bestaat uit drie delen: *talen*, *tools* en *case-studies*.

Het eerste gedeelte van dit proefschrift introduceert de talen die een rol spelen bij de specificatie van parallelle systemen. In hoofdstuk 1 wordt de specificatietaal PSF beschreven, die in het gehele proefschrift wordt gebruikt om specificaties op het voor mensen begrijpelijke niveau te beschrijven. De verschillende taalconstructies worden stap voor stap geïntroduceerd en aan de hand van een doorlopend voorbeeld toegelicht. Dit voorbeeld wordt steeds complexer naarmate meer taalconstructies worden gebruikt. Naast de syntax wordt ook de semantiek van elke taalconstructie gegeven.

Hoewel PSF een geschikte taal is voor mensen om het gedrag van parallelle systemen uit te drukken, is het niet de meest optimale taal om als *interface* tussen de verschillende tools te dienen. Een verzameling tools kan veel baat hebben bij een gelaagd ontwerp, waarbij verschillende talen worden gebruikt voor de verschillende abstractieniveau's. De taal TIL (*Tool Interface Language*), die gebruikt wordt voor de communicatie tussen de verschillende tools in de PSF Toolkit onderling, wordt beschreven in hoofdstuk 2.

Hoofdstuk 3 behandelt de vertaling van een PSF-specificatie naar een TIL-specificatie. Naast het feit dat expressies van de ene syntax in de andere worden omgezet, moet ook nog een andere transformatie toegepast worden, namelijk het verwijderen van modulaire constructies, ook wel *normalisatie* genoemd. PSF ondersteunt het gebruik van modulen om verwante delen van een specificatie te groeperen tot een hiërarchisch ontwerp. Het module-concept heeft voornamelijk tot doel om grote specificaties op een systematische manier te kunnen indelen, zodat men niet het overzicht verliest. Op het niveau van de tools heeft zo'n indeling in modules echter geen functie en wordt de voorkeur gegeven aan één grote specificatie zonder extra structureringsmechanismen.

Het tweede deel van dit proefschrift concentreert zich op de tools in de PSF Toolkit. Hoofdstuk 4 is gewijd aan de *simulator*, een tool dat het gedrag van een specificatie kan simuleren. De simulator is een interactief tool, hetgeen wil zeggen, dat de gebruiker het gedrag van het systeem kan beïnvloeden door tijdens een sessie verschillende externe boodschappen (*events*) aan de simulator aan te bieden. Hoofdstuk 4 begint met een beschrijving van de kern van het algoritmische gedeelte van de simulator, namelijk het berekenen van de *head normal form* van een proces-expressie. Deze beschrijving wordt gegeven door middel van een algebraïsche specificatie geschreven in PSF zelf. Vervolgens worden de implementatie van de simulator en zijn gebruikersinterface beschreven.

Hoofdstuk 5 behandelt de *proof assistant* uit de PSF Toolkit. De proof assistant is een tool dat het mogelijk maakt om bewijzen te construeren door stapsgewijze transformaties van proces-expressies. De belangrijkste eigenschap van dit tool is dat het in staat is de gebruiker een keuze te laten maken uit de, volgens de onderliggende semantiek voor proces-operatoren, mogelijke bewijsstappen. De proof assistant omvat bovendien zogenaamde *tactics*, combinaties van veel voorkomende bewijsstappen die als eenheid uitgevoerd worden. Naast de implementatie en de gebruikersinterface wordt het gebruik van de proof assistant geïllustreerd aan de hand van een voorbeeld.

Hoofdstuk 6 beschrijft kort de overige tools in de PSF Toolkit, waaronder de *termherschrijver*, de *compiler driver* en *library manager*, de *initiële-algebra-generator*, de *transitiesysteem-generator* en de *equivalentie-tester*.

Het laatste deel van dit proefschrift behandelt twee case-studies die werden uitgevoerd met behulp van PSF en de PSF Toolkit. De eerste case-study, beschreven in hoofdstuk 7, is het resultaat van een courseware project in het kader van het college Software Engineering. Het doel van dit project was om een zogenaamd *software-harnas* aan te bieden aan de studenten die tijdens het practicum de opdracht kregen om een algebraïsche specificatie van een compiler voor een simpele programmeertaal te schrijven. In hoofdstuk 8 wordt een specificatie gegeven van een gedeelte van de GKS (*Graphics Kernel System*) ISO standaard. Hierbij werd uitgegaan van een bestaande specificatie in CSP. Dit voorbeeld laat zien dat de beschikbaarheid van een verzameling tools van groot belang is voor het controleren van zowel de syntax als de correctheid van een specificatie.

Tenslotte besluit hoofdstuk 9 dit proefschrift met een evaluatie van de ontwikkeling van PSF en de PSF Toolkit. Hier worden de sterke en de zwakke punten van het ontwerp en de implementatie besproken, en tevens suggesties gedaan voor mogelijke verbeteringen en uitbreidingen.

REFERENCES

[AB91] G.J. Akkerman & J.C.M. Baeten, *Term rewriting analysis in process algebra*, CWI Quarterly 4 (4), 1991, pp, 257-267, CWI, 1991.

[Arb92] F. Arbab, *Specification of Manifold*, Report CS-R9220, CWI, Amsterdam 1991.

[Bae90] J.C.M. Baeten (ed.), *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[Bar90] D. Barrow, *From M-TIL to TIL*, Master's thesis, Programming Research Group, University of Amsterdam, 1990.

[BB88] J.C.M. Baeten & J.A. Bergstra, *Global renaming operators in concrete process algebra*, Information & Computation, 78, pp. 205-245, 1988.

[BB91] J.C.M. Baeten & J.A. Bergstra, *Real time process algebra*, Formal Aspects of Computing, 3 (2), 1991, pp. 142-188, 1991.

[BBK86] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra*, Fundamenta Informaticae IX (1986), pp. 127-168, IOS Press, 1986.

[BBK87] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Conditional axioms and α/β-calculus in process algebra*, in: Proceedings IFIP Conference on Formal Description of Programming Concepts III, Ebberup, (M. Wirsing, ed.) pp. 77-103, North-Holland, 1987.

[BBMV91] J.C.M. Baeten, J.A. Bergstra, S. Mauw & G.J. Veltink, *A process specification formalism based on static COLD*, in: Algebraic Methods II: Theory, Tools and Applications, (J.A. Bergstra & L.M.G. Feijs, eds.), LNCS 490, pp. 303-335, Springer Verlag, 1991.

[BHK89] J.A. Bergstra, J. Heering & P. Klint, *The algebraic specification formalism ASF*, in: *Algebraic specification*, J.A. Bergstra, J. Heering & P. Klint (eds.), pp. 1-66, ACM Press Frontier Series, Addison-Wesley 1989.

[BHK90] J.A. Bergstra, J. Heering & P. Klint, *Module Algebra*, Journal of the ACM, 37(2), pp. 335-372, 1990.

[BK84] J.A. Bergstra & J.W. Klop, *Process algebra for synchronous communication*, Information & Control 60, pp. 109-137, 1984.

[Bou91] L.G. Bouma, *Algebraische Specificaties*, Kluwer, 1991.

[BP93] J.J. Brunekreef & A. Ponse, *An algebraic specification of a model factory, part IV*, Report P9316, Programming Research Group, University of Amsterdam, 1993.

[BW89] L.G. Bouma & H.R. Walters, *Implementing algebraic specifications*, in: *Algebraic specification*, J.A. Bergstra, J. Heering & P. Klint (eds.), pp. 199-282, ACM Press Frontier Series, Addison-Wesley 1989.

[BW90] J.C.M. Baeten & W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[CCITT88] CCITT *Specification and Description Language SDL*, Recommendation Z.100, CCITT Blue Book, 1988.

[Cle90] R. Cleaveland, *On automatically distinguishing inequivalent processes* , in: Proceedings CAV'90, E.M. Clarke & R.P. Kurshan (eds.), pp. 463-477, AMS, 1990.

[CLV93] S.T. Chanson, A.A.F. Loureiro & S.T. Vuong, *On tools supporting the use of formal description techniques in protocol development,* Computer Networks and ISDN Systems 25 (1993), pp. 723-739, Elsevier Science Publishers, 1993.

[CM81] W.F. Clocksin & C.S. Mellish, *Programming in Prolog*, Springer Verlag, 1981.

[CPS89] R. Cleaveland, J. Parrow & B. Steffen, *The Concurrency Workbench*, in: Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems, J. Sifakis (ed.), LNCS 407, pp. 24-37, Springer Verlag, 1989.

[Cro91] F. Croes, *Transformation of a PSF specification into a finite transition system*, Master's thesis, Programming Research Group, University of Amsterdam, 1991.

[Die94] B. Diertens, *New Features in PSF I: Interrupts, Disrupts, and Priorities*, Report P9417, Programming Research Group, University of Amsterdam, 1994.

[Dik89] C.H.S. Dik, *A Fast Implementation of the Algebraic Specification Formalism,* Master's Thesis, University of Amsterdam, 1989.

[DJ90] N. Dershowitz & J.P. Jouannaud, *Rewrite Systems*, in: Handbook of Theoretical Computer Science, Formal Models and Semantics, (J. van Leeuwen, ed.), pp. 243-320, North Holland, 1990.

[DLH89] D.A. Duce, R. van Liere & P.J.W. ten Hagen, *Components, frameworks and GKS input*, in: Eurographics '89 Conference Proceedings, W. Hansmann, F.R.A. Hopgood & W. Straßer (eds.), North Holland, 1989.

[DP94] B. Diertens & A. Ponse, *New Features in PSF II: Iteration and nesting*, Report P9425, Programming Research Group, University of Amsterdam, 1994.

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

[FJ93] L. Feijs & H. Jonkers, *Formal Specification and Design*, Cambridge Tracts in Theoretical Computer Science 35, Cambridge University Press, 1993.

[FM91] J.C. Fernandez & L. Mounier, *A tool set for deciding behavioral equivalences,* in: Proceedings CONCUR '91, Amsterdam, J.C.M. Baeten & J.F. Groote (eds.), LNCS 527, pp. 23-42, Springer Verlag, 1991.

[FMdS91] J.C. Fernandez, E. Madeleine & R. de Simone, *FC: A common format representation for Automata, Version 2*, in: Deliverables ESPRIT Basic Research Action No. 3006: CONCUR, Task 3.2.3, 1991.

[GP90] J.F. Groote & A. Ponse, *The syntax and semantics of μCRL*, Report CS-R9076, CWI, Amsterdam 1990.

[GP91] J.F. Groote & A. Ponse, *Proof theory for μCRL*, Report CS-R9138, CWI, Amsterdam 1991.

[GS90] H. Garavel & J. Sifakis, *Compilation and Verification of LOTOS Specifications*, in: Protocol Specification, Testing and Verification, X, (L. Logrippo, R.L. Probert & H. Ural, eds.) pp. 379-394, North-Holland, 1990.

[GV90] J.F. Groote & F.W. Vaandrager, *An efficient algorithm for branching bisimulation and stuttering equivalence*, in: Proceedings 17th ICALP, Warwick, (M.S. Paterson, ed.) LNCS 443, pp. 626-638, Springer Verlag, 1990.

[GV93] R.J. van Glabbeek & F.W. Vaandrager, *Modular specifications in process algebra*, TCS 113(2), pp. 293-348, 1993.

[GW89] R.J. van Glabbeek & W.P. Weijland, *Branching time and abstraction in bisimulation semantics (extended abstract)*, in: Information Processing 89 (G.X. Ritter, ed.), pp. 613-618, Elsevier Science Publishers (North Holland),1989.

[Hen88] 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.

[Hen91] P.R.H. Hendriks, *Implementation of Modular Algebraic Specifications*, Ph.D. Thesis, University of Amsterdam, 1991.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint & J. Rekers, *The syntax definition formalism SDF - reference manual*, SIGPLAN Notices, 24(11), pp. 43-75, 1989.

[Hil87] M. Hillerström, *Verification of CCS-processes*, M.Sc. Thesis, Computer Science Department, Aalborg University, 1987.

[HK89] J. Heering & P. Klint, *The syntax definition formalism SDF*, in: *Algebraic specification*, J.A. Bergstra, J. Heering & P. Klint (eds.), pp. 283-298, ACM Press Frontier Series, Addison-Wesley 1989.

[HM85] M. Hennessy & R. Milner, *Algebraic Laws for Nondeterminism and Concurrency*, Journal of the Association for Computing Machinery, vol. 32, nr. 1, pp. 137-161, 1985.

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

[HP92] J.A. Hillebrand & A. Ponse, *An algebraic specification of a model factory, part II*, Report P9214, Programming Research Group, University of Amsterdam, 1992.

[HU79] J.E. Hopcroft & J.D. Ullman, *Introduction to Automata Theory, Languages and Computation,* Addison-Wesley, 1979.

[INMOS88] INMOS Limited, *occam® 2 Reference Manual*, Prentice Hall, 1988.

[ISO85] ISO, *Information processing systems - Computer Graphics - Graphical Kernel System (GKS) functional description*, ISO IS7942, 1985.

[ISO89a] International Organization for Standardization, *IS 8807, Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 1989.

[ISO89b] International Organization for Standardization, *IS 9074, Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique Based on an Extended State Transition Model*, ISO/IEC 1989.

[Jac89] H. Jacobsson, *Proposal for TIL*, Master's thesis, Programming Research Group, University of Amsterdam, 1989.

[Jac92] I. Jacobson, *Object Oriented Software Engineering*, Addison-Wesley, 1992.

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

[Kap87] S. Kaplan, *A Compiler for Conditional Term Rewriting Systems*, in: Rewriting Techniques and Applications, (P. Lescanne, ed.), LNCS 256, pp. 25-41, Springer Verlag, 1987.

[KBG93] G. Karjoth, C. Binding & J. Gustafsson, *LOEWE: A LOTOS engineering workbench,* Computer Networks and ISDN Systems 25 (1993), pp. 853-874, Elsevier Science Publishers, 1993.

[Kli91] P. Klint, *A meta-environment for generating programming environments*, in: Algebraic Methods II: Theory, Tools and Applications, (J.A. Bergstra & L.M.G. Feijs, eds.), LNCS 490, pp. 105-124, Springer Verlag, 1991.

[Klo91] J.W. Klop, *Term Rewriting Systems*, in: Handbook of Logic in Computer Science, Volume II, (S. Abramsky et al. eds.), Oxford University Press, Oxford, 1991.

[Koo94] J.W.C. Koorn, *Generating uniform user-interfaces for interactive programming environments,* Ph.D. Thesis, Programming Research Group, University of Amsterdam, 1994.

[Kor92] H.P. Korver, *Computing distinguishing formulae for branching bisimulation,* in: Proc. Third Workshop on Computer Aided Verification (CAV '91), Aalborg 1991, (K.G. Larsen & A. Skou, eds.), LNCS 575, pp. 13-23, Springer Verlag, 1992.

[Kor94] H.P. Korver, *A Theory for Simulators*, in: The Computer Journal, 4, vol. 37, pp. 279-287, 1994.

[KR88] B.W. Kernighan & D.M. Ritchie, *The C Programming Language*, second edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[KV85] C.P.J. Koymans & J.L.M. Vrancken, *Extending process algebra with the empty process ε*, report LGPS 1, Dept. of Phil.,State University of Utrecht, 1985.

[Lin92] H. Lin, *PAM: A Process Algebra Manipulator*, in: Proc. Third Workshop on Computer Aided Verification (CAV '91), Aalborg 1991, (K.G. Larsen & A. Skou, eds.), LNCS 575, pp. 136-146, Springer Verlag, 1992.

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

[Mau89] S. Mauw, *An algebraic specification of process algebra, including two examples*, in: Algebraic Methods: Theory, Tools and Applications, Passau 1989, (M. Wirsing & J.A. Bergstra, eds.), LNCS 394, pp. 507-554, Springer Verlag, 1989.

[Mau91] S. Mauw, *PSF, A process Specification Formalism*, Ph.D. Thesis, Programming Research Group, University of Amsterdam, 1991.

[MdSV89] E. Madeleine, R. de Simone & D. Vergamini, *ECRINS V2-1, A Proof Laboratory for Process Calculi*, INRIA, 1989.

[Mey88a] B. Meyer, *Object Oriented Software Construction*, Prentice-Hall, 1988.

[Mey88b] B. Meyer, *Eiffel: A Language and Environment for Software Engineering*, in: The Journal of Systems and Software, 1988.

[Mil80] R. Milner, *A calculus of communicating systems*, Springer LNCS 92, 1980.

[Mul90] J.C. Mulder, *Case studies in process specification and verification*, Ph.D. Thesis, Programming Research Group, University of Amsterdam, 1990.

[MV89a] 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.

[MV89b] S. Mauw & G.J. Veltink, *A Tool Interface Language for PSF*, Report P8912, Programming Research Group, University of Amsterdam, 1989.

[MV90] S. Mauw & G.J. Veltink, *A process specification formalism*, Fundamenta Informaticae XIII (1990), pp. 85-139, IOS Press, 1990.

[MV92] S. Mauw & G.J. Veltink, *A proof assistant for PSF*, in: Proc. Third Workshop on Computer Aided Verification (CAV '91), Aalborg 1991, (K.G. Larsen & A. Skou, eds.), LNCS 575, pp. 158-168, Springer Verlag, 1992.

[MV93] S. Mauw & G.J. Veltink (eds.), *Algebraic Specification of Communication Protocols*, Cambridge Tracts in Theoretical Computer Science 36, Cambridge University Press, 1993.

[ODo85] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press, 1985.

[Oss77] J.F. Ossanna, *TROFF User's Manual*, Technical Report 54, Bell Laboratories Computing Science, 1977.

[Par81] D.M.R. Park, *Concurrency and automata on infinite sequences*, in: Proc. 5th GI Conference, (P. Deussen, ed.), LNCS 104, pp. 167-183, Springer Verlag, 1981.

[Pet80] C. Petri, *Introduction to the general net theory*, in: Net Theory and Applications, (W. Brauer, ed.), LNCS 84, pp. 1-19, Springer Verlag, 1980.

[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.

[Pol92] E.E. Polak, *An efficient implementation of branching bisimulation and distinguishing formulae*, Report P9216, Programming Research Group, University of Amsterdam, 1992.

[PV93] A. Ponse & J.A. Verschuren, *An algebraic specification of a model factory, part III*, Report P9303, Programming Research Group, University of Amsterdam, 1993.

[RdS91] V. Roy & R. de Simone, *An Autograph Primer (Version 3)*, INRIA, 1991.

[Rei85] W. Reisig, *Petri Nets*, EATCS monograph on TCS, Springer Verlag, 1985.

[Rum91] J. Rumbaugh, et al., *Object Oriented Modeling and Design*, Prentice-Hall, 1991.

[SAHH91] D. Soede, F. Arbab, I. Herman & P.J.W. ten Hagen, *The GKS Input Model in MANIFOLD*, in: Computer Graphics Forum 10, pp. 209-224, North-Holland, 1991.

[SPECS89] SPECS, *Definition of MR and CRL Version 2.0*, Deliverable WP5.4, RACE project 1046, The SPECS Consortium, 1989.

[Str86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[TT94] B.C. Thompson & J.V. Tucker, *Equational specification of synchronous concurrent algorithms and architectures (Second Edition)*, Department of Computer Science Report 15-94, Unviersity of Wales, Swansea, 1994.

[vDel91] A. van Delft, *The Scriptic Programming Language*, in: Parle '91 volume II: Parallel Languages, Eindhoven, E.H.L. Aarts, J. van Leeuwen & M. Rem (eds.), LNCS 506, pp. 220-237, Springer Verlag, 1991.

[vEij92] P. van Eijk, *The Lotosphere Integrated Tool Environment: LITE*, in: Proc. Formal Description Techniques IV, IFIP TC6/WG6.1, FORTE '91, Sydney, (K.R. Parker & G.A. Rose, eds.), pp. 471-475, North-Holland, 1992.

[Vel90] G.J. Veltink, *From PSF to TIL*, Report P9009, Programming Research Group, University of Amsterdam, 1990.

[Vel92] G.J. Veltink, *XP, an experiment in modular specification*, in: Proc. Formal Description Techniques IV, IFIP TC6/WG6.1, FORTE '91, Sydney, K.R. Parker & G.A. Rose (eds.), pp. 149-164, North-Holland, 1992.

[Vel93] G.J. Veltink, *The PSF Toolkit*, Computer Networks and ISDN Systems 25 (1993), pp. 875-898, Elsevier Science Publishers, 1993.

[Ver91] D. Vergamini, *Auto/Mauto Users' Manual*, INRIA Research Report 111, INRIA, 1991.

[Ver92a] C. Verhoef, *Linear unary operators in Process Algebra*, Ph.D. Thesis, Programming Research Group, University of Amsterdam, 1992.

[Ver92b] J.A. Verschuren, *A Simulator for μCRL in ASF+SDF*, Report P9203, Programming Research Group, University of Amsterdam, 1992.

[VW92] S.F.M. van Vlijmen & A. van Waveren, *An algebraic specification of a model factory*, Report P9209, Programming Research Group, University of Amsterdam, 1992.

[VW93] S.F.M. van Vlijmen & A. van Waveren, *Algebraic Specification of a System for Traffic Regulation at Signalized Intersections*, Report P9313, Programming Research Group, University of Amsterdam, 1993.

[Wal91] H.R. Walters, *On Equal Terms. Implementing Algebraic Specifications*, Ph.D. Thesis, Programming Research Group, University of Amsterdam, 1991.

[Wam93] J.J. van Wamel, *A library for PSF*, Report P9301, Programming Research Group, University of Amsterdam, 1993.

[Wie91] F. Wiedijk, *Persistence in Algebraic Specifications*, Ph.D. Thesis, Programming Research Group, University of Amsterdam, 1991.

[Wir71] N. Wirth, *The Programming Language Pascal*, Acta Informatica 1 (1971) , pp. 35-63, 1971.

Tools for PSF

G.J. Veltink

This book addresses the design and the implementation of software tools for the specification language PSF (Process Specification Formalism).

The language PSF is described as well as the tools that form the PSF Toolkit.

Two case studies showing the usage of PSF and the PSF Toolkit are included.

ISBN 90-74795-29-3

STELLINGEN

behorende bij het proefschrift

Tools for PSF

G.J. Veltink

1 Het specificeren van grote en complexe problemen in een algebraïsche specificatietaal is en blijft een soort van programmeren, met inbegrip van alle nare gevolgen, zoals 'debuggen'.

2 Het vervangen van de algemene recursie in ACP door iteratie (Kleene's binary star operator) kan de implementatie van een aantal tools voor deze taal aanmerkelijk vereenvoudigen.

3 Als een specificatie zou moeten dienen als een document dat de eisen aan een te implementeren systeem vastlegd, moeten zowel de opdrachtgever als de opdrachtnemer dit document begrijpen en op dezelfde manier interpreteren.

In de realiteit wordt een specificatie meestal na een aantal kleine wijzigingen geaccepteerd en is de opdrachtgever echter dikwijls pas in staat een systeem werkelijk te beoordelen als een prototype, of erger nog de complete implementatie, afgeleverd wordt.

4 Om tot een grotere acceptatie te komen, zullen specificaties geschreven in specificatietalen die tot doel hebben executeerbare systemen te specificeren, ook machinaal in executeerbare code omgezet moeten kunnen worden.

Indien dit niet mogelijk is en een specificatie 'met de hand' omgezet moet worden, loopt men het gevaar dat bij latere veranderingen niet beide documenten gelijktijdig worden aangepast. Een dergelijk probleem is momenteel dikwijls als de discrepantie tussen implementatie en documentatie te bespeuren.

5 Het verdient de voorkeur in een formele specificatie een datatype te benoemen naar een element uit dit datatype, zodat bijvoorbeeld de definitie "$x : INTEGER$" gelezen kan worden als "x is een $INTEGER$" of "x is van het type $INTEGER$", dit in tegenstelling tot: "$x : INTEGERS$" met de lezing "x is een element uit de $INTEGERS$".

In de programmeertaal Chill moet een enumerated type met behulp van het keyword SET gedefinieerd worden. Dit heeft tot gevolg dat voor een werkelijke verzameling (van enumerations) noodgedwongen het keyword POWERSET gebruikt moet worden.

6 Elke letterlijke numerieke constante in de source code van een computerprogramma anders dan *0* of *1* is bij voorbaat verdacht en een mogelijke bron van fouten.

7 De opkomst van zogenaamde desktop-publishing programmatuur heeft er weliswaar toe geleid dat (professioneel) drukwerk met minder middelen geproduceerd kan worden, echter, daar de gestelde verwachtingen aan drukwerk ook gestegen zijn, is de benodigde tijd om tot een acceptabel resultaat te komen minstens gelijk gebleven, zo niet toegenomen.

8 Het is opmerkelijk dat QDOS, het operating system van de Sinclair QL home computer, in 1983 reeds pre-emptive multitasking aanbood, iets waar gebruikers van MS-Windows of MacOS anno 1995 nog steeds reikhalzend op zitten te wachten.

9 De voortschrijdende digitalisering binnen de communicatietechnologie maakt het, technisch gezien, mogelijk steeds meer diensten aan te bieden waarvan niet in te schatten is of er daadwerkelijk behoefte aan is.

Het is bijvoorbeeld, gezien het huidige marktaandeel van 'Pay-TV', vooralsnog onduidelijk of er ooit een commercieel voldoende grote 'demand' naar 'video on demand'-systemen zal ontstaan

10 Het begrip 'information hiding' wordt in de informatica jammer genoeg dikwijls op de verkeerde plaats toegepast, namelijk bij de documentatie.

"Information Hiding hat nichts mit Handbüchern zu tun!"
Stefan Richter 3/6/93.