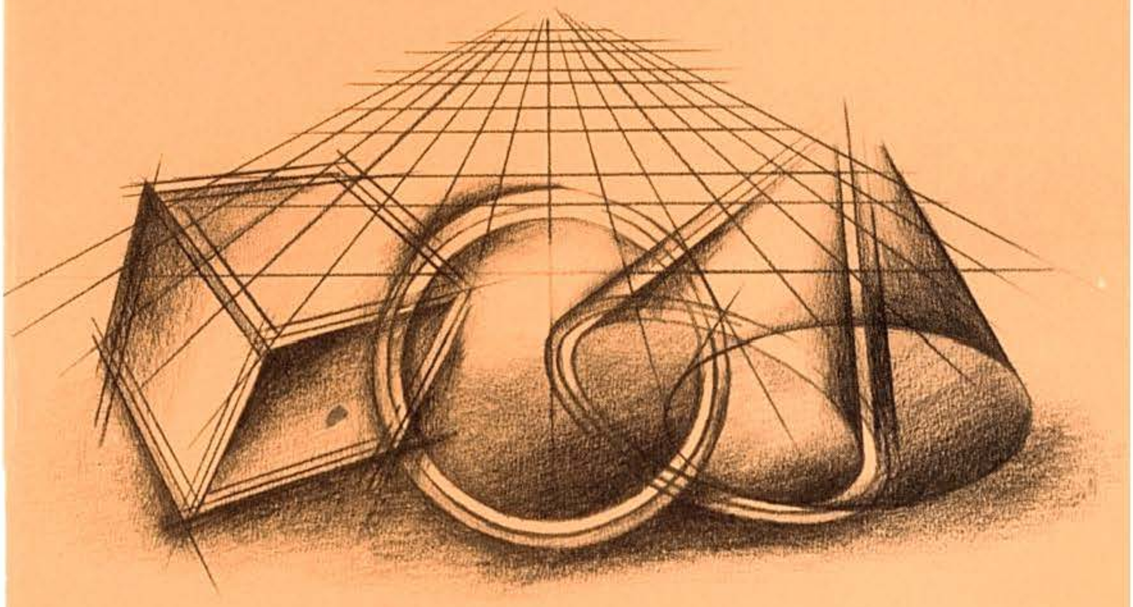
PSF

A Process Specification Formalism



Sjouke Mauw

PSF - A Process Specification Formalism

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
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op woensdag 18 december 1991 te 10.30 uur

door
Sjouke Mauw

geboren te Amsterdam

Vakgroep Programmatuur
Universiteit van Amsterdam
1991

Promotor:	prof.dr. J.A. Bergstra
Co-promotor:	prof.dr. J.C.M. Baeten
Faculteit:	Wiskunde en Informatica

This work was partially sponsored by ESPRIT contract no. 432, An integrated formal approach to industrial software development (METEOR) and by ESPRIT BRA contract 3006, Theories of concurrency: Extension and Unification (CONCUR).
Chapter 2 is reproduced with permission of Elsevier Science Publishers, chapter 5 is reproduced with permission of Cambridge University Press and chapter 6 is reproduced with permission of Springer Verlag.

voor Marjan

PREFACE

Since 1985, when I started my research in the field of algebraic techniques for software specification, I developed the ambition to make process algebra operational. In the process algebra group at the University of Amsterdam and the Center for Mathematics and Computer Science (CWI), this was an unexplored subject. A first attempt was to catch process algebra in an algebraic specification formalism. This worked out quite well, but the resulting specification was not executable (see [82]). This was mainly a consequence of the undecidable character of most notions in process algebra.

In spite of this undecidability, it seemed that software development using process algebra could benefit from a series of tools that could be an aid in specification, simulation, verification and implementation, or even automate it. The writing of a simple prototype of a process simulation tool made it clear to me that the first thing to do was the design of an input language for such a tool. This language was called PSF, and now it is the basis of all implementation directed work on process algebra in our group.

This specification language, together with a number of implementation issues and case studies was the topic of the research of which this thesis reports. The finishing of this thesis does not imply that the work on the toolkit and related aspects is also finished. It just provides a snapshot of a project that is still going on.

ACKNOWLEDGEMENTS

Of the various people who influenced the contents and preparation of this thesis, my promotor Jan Bergstra and co-promotor Jos Baeten take a special position. Jan, with his symbiotic view on the combination of science and management created an encouraging environment for scientific research. His optimism and richness of ideas stimulated me a lot in the preparation of this thesis.

I thank Jos for his friendly though critical guidance. His comments were always very thorough and useful. He showed a good eye for detail by reading and debugging the boring proofs and lengthy specifications I came up with.

Parts of this thesis were written in fruitful co-operation with Gert Veltink and Freek Wiedijk. I thank Gert for his efficient and constructive way of working and Freek for sharing both his enthusiasm and contempt with respect to computer science. His presence as well as his regular absence stimulated me in writing this thesis.

Thanks are due to Hans Mulder because he never failed in pointing out the weakness in the algebraic approach towards software specification. He eagerly showed his knowledge of computer operation and skillfulness in close reading by finding bugs in computer programs and in papers.

At the weekly Process Algebra Meetings I always enjoyed the open atmosphere of critical co-operation. Internal reviews and brain-storms as at this seminar are necessary conditions to keep the work of a group of researchers focussed and up to date. Therefore I thank the participants of the Process Algebra Meetings, among whom Wiet Bouma, Nicolien Drost, Rob van Glabbeek, Henk Goeman, Jan Friso Groote, Jan Willem Klop, Karst Koymans, Alban Ponse, Gerard Renardel, Piet Rodenburg, Frits Vaandrager, Chris Verhoef, Jos Vrancken, Fer-Jan de Vries, Peter Weijland and Han Zuidweg.

I thank the members of the GIPE group for their assistance in using ASF and SDF: Paul Hendriks, Emma van der Meulen, Jan Rekers, Pum Walters and especially Paul Klint whose suggestions on practical issues were of great value and Wilco Koorn for programming a prototype of the PSF type checker.

I thank Casper Dik for keeping the machines running, Bob Diertens for his skillful programming and Madelon Drolsbach for secretarial assistance.

Furthermore a number of master's students from the University of Amsterdam helped by performing case studies and writing software: Firoez Azarhoosh, Duncan Barrow, Jeroen Brouwer, Felix Croes, Henrik Jacobsson, and Arnout Oldenburger.

I thank the promotion committee, consisting of Jan Bergstra, Jos Baeten, Paul Klint, Scott Smolka, Martin Rem and Karst Koymans, for their comments.

Finally, my special thanks go to Hans van Herwijnen who designed the cover.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

1. SPECIFICATION

The recognition that in the construction of software a number of different phases can be distinguished, led to the formulation of the classical life cycle model for software development, as described e.g. by Sommerville [97]. In this model, the actual coding of the software only starts after a thorough analysis of the problems involved and a design of the solutions for these problems. The result of this design phase is the specification, a complete description of the way in which the problems are to be solved.

For this purpose, several specification techniques have been developed. These techniques range from the use of (structured) natural languages and informal drawings, to formal specification techniques. Often the resulting computer program itself is considered as the specification of the system. The main problem with informal specification techniques is that they allow ambiguities, inconsistencies and incompleteness, while on the other hand a formal approach enables formal verification. These are the main reasons that there is an increased interest in formal specification techniques by software developers. If the specification technique allows to write executable specifications, it can be used as a method for rapid prototyping. This type of software development is emphasized in the spiral model of Boehm [31].

Formal techniques comprise wide spectrum languages, such as COLD [41] and VDM [102], as well as dedicated techniques for small domains, such as SDL [44] for telecommunication applications and LOTOS [67] for concurrent systems. The formal description technique described in this thesis is designed for the domain of concurrent, or parallel, systems.

2. CONCURRENT SYSTEMS

Due to their non-deterministic nature, concurrent systems are even harder to program correctly than sequential ones. Since testing and debugging concurrent systems is hard, the need for formal verification and thus formal techniques becomes clear. The occurrence of a deadlock, for example, may be difficult to reproduce, and thus the source of such an error might be difficult to localize. Only a formal verification can ensure deadlock freedom for all possible scenarios.

A number of formal techniques for parallel systems are gathered in the term process algebra. The main members of this family are CCS [90], CSP [61] and ACP [22]. ACP, the Algebra of Communicating Processes will be the starting point for this thesis. It allows one to specify a concurrent system algebraically and to produce an algebraic verification of correctness.

3. PSF

ACP has been used in practice for a number of relatively small case studies (as in [4]). Already in these simple examples the need was felt for computer support in the specification and verification process. This resulted in the development of a number of ad-hoc computer-tools which remained in the prototype phase.

The first step towards the construction of an integrated toolset for ACP is described in this thesis. That is the description of an ACP based specification language with a fixed syntax and semantics, named PSF (Process Specification Formalism).

A number of reasons prevents the use of ACP itself as a specification language. First of all, the syntax of ACP is not fixed and contains non-ASCII symbols. The second reason is that the data objects involved in ACP specifications have no formal status. The third reason is that in order to produce more complex and extensive examples, a mechanism for structuring specifications is needed.

The specification language PSF, as described in chapter 2 of this thesis, deals with these shortcomings. PSF has a fixed syntax in which, for example, the sets and communication function occurring in a specification have an explicit definition, rather than an informal description. Furthermore, data objects in PSF are defined using the technique of algebraic specifications and finally, PSF supports modularization and parameterization of specifications. Beside a motivation for the PSF language, this chapter also contains a comparison of PSF with other languages for parallelism.

Since the extension of ACP with new operators and constants is being studied comprehensively, the choice had to be made of what features to include in PSF. We chose to include only the basic and well understood ACP constants and operators. The practical use of PSF will indicate what extensions are to be incorporated in a possible update of PSF. The issue of extending PSF is discussed in chapter 3 of this thesis.

4. Towards Tools

When actually building tools for ACP, it is desirable to make the tools independent of the actual specification language as much as possible. This reduces the risks related to a possible misdesign of the language. Another reason why PSF itself should not be used as the core language of the toolkit is that, although features such as modularization, parameterization and overloading are useful and even necessary for writing specifications, they do not allow for easy parsing and manipulation by computer-tools. Furthermore, the process of normalizing a specification, that is removing all modular structure, is quite costly (see [103]).

These observations led to the design of a format for the internal representation of specifications. This Tool Interface Language (TIL) is described in chapter 4. The free format feature of TIL provides the tools with a means to insert their own tool dependent information in order to communicate it to other tools.

TIL also allows for input from other sources or other specification languages for parallel systems.

The current status of the toolkit is described in [104]. It contains a parser and a syntax and type checker for PSF, a compiler from PSF to TIL which makes use of a library manager, a term-rewriting tool, a simulation tool and a proof assistant (as described in [88]). The work on other tools is continuing, such as a tool for deciding bisimulation equivalence for regular processes, based on work from [53] and [81].

5. Case Studies

Application of PSF to practical case studies is a test for its usefulness. The purpose of case studies is to provide insight in the possibilities and limitations of the formalism. Not only concerning the expressiveness and the concrete syntax, but also concerning the role that the formalism may play in the specification phase of the life cycle model. This means that we investigate if it is possible to provide a functional specification starting with a requirements specification in either a formal or an informal language, if it is suited for formal verification and validation and if it is a clear starting point for implementation.

The case studies provided in this thesis are of two hypothetical systems, a computer integrated manufacturing system (chapter 5) and a transit node (chapter 6). The CIM case study is based on a specification in the language LOTOS [30] and is supplied with a verification in ACP of the correctness of the specification. The case study of the transit node is an exercise to design a specification based on a requirement study which was both formally and informally expressed.

Other case studies in the PSF language are in [69], [84], [91] and [33].

6. PRELIMINARIES

This thesis will not contain a detailed introduction to the theory of ACP. Chapter 2 contains a quick overview. The textbook [14] gives a thorough treatment of all relevant concepts. Other overview papers on ACP are [23] and [22].

We will not provide a formal proof-theory for ACP with abstract data types. This means that, although PSF has a formal operational semantics, verification of PSF specifications are still in an informal ACP-like format. A formal proof-theory for the PSF related language μ-CRL is developed in [52].

Also the theory of algebraic specification is not elaborated extensively. Refer to [40] or [20] for more details on this subject. Chapter 2 contains a short introduction to the algebraic specification language ASF and the syntax definition formalism SDF [56].

7. ORGANIZATION OF THIS THESIS

This thesis, consisting of a number of papers, can be divided into three parts. The first part deals with the definition of the specification language PSF (in some papers referred to as PSF_d). Chapter 2 (appeared as [87]) contains the description of the syntax and semantics of PSF and demonstrates the use of the language by a number of simple examples. It concludes with a comparison of PSF with related languages. An introduction to PSF is presented in [85]. Chapter 3 considers possible extensions of PSF. The need for a stable language somehow contradicts the practice to enhance ACP with some operators needed for a special application. Guide-lines and examples are given on how to enhance PSF without making all existing tools and specifications obsolete. The extensions considered include conditional choices, operators for disabling, interrupts and priorities, and constructions for state manipulation.

The second part (chapter 4) describes an implementation directed issue. It contains the definition of the Tool Interface Language TIL, which appeared as [86]. It discusses the criteria for the design of TIL and its syntax and semantics.

The third part comprises two case studies. The first case study (chapter 5) contains a specification of two CIM-architectures, which is a revised version of [83]. This case study contains two protocols for co-operating machines in a factory environment, which are both proven correct. The first protocol is based on [30]. The second protocol is derived from the first one by adding more capabilities. The specification of the transit node in chapter 6 was published as [89]. It is the common case study in the METEOR project. The transit node is a hypothetical device (or cluster of devices) which can be used to route data from one location to another. The specification is based on a requirements analysis in the ERAE language [55] and contains a description of the method which lead to this specification.

All complete specifications in this thesis have been checked with the PSF and SDF type checkers.

Chapter 2

A PROCESS SPECIFICATION FORMALISM

(with G.J. Veltink)

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

1. MOTIVATION

The last decade has shown yet another revolution in computer technology: parallel computers. And with this progression in hardware ought to come a progression in the development of software. Though software development has come a long way, since the days that Alan Turing fed machine code programs written in Baudot code into the Colossus [62], through assemblers, higher level languages like FORTRAN and later Pascal and even Ada†, it still finds itself in the midst of the so-called *software crisis*. With the introduction of parallel computers things tend to become even worse.

1.1. PROBLEMS WITH SOFTWARE

It seems that developers of software were not yet ready for this next step. It is a well-known fact that large programs, though being used in every day services, have the nasty property of still containing software errors or *bugs* as they are called in the vernacular. That programmers have become used to this fact is evidenced by the UNIX‡ reference manuals [16], wherein, in the standard format for describing programs, there is a special section devoted to the known bugs. Construction of large programs may be difficult, but maintaining and tailoring programs to new needs is far more complex. In many cases it is better to rewrite the whole program from scratch than to try and adapt it.

1.2. BUILDING BETTER PROGRAMS

Of course there have been attempts to develop methods to help programmers in constructing software and excluding errors. One of the most formal approaches, using mathematical techniques, is the proving of program correctness. See for example the work by Dijkstra [37] and Hoare [60]. Though these formal methods do well for small programs, they have never really found their way to the programming-in-the-large. Other methods use data flow charts and different kinds of graphical representations of the program to help the programmer.

Yet another method to support program development is the use of *programming environments*. A programming environment consists of a large number of tools and an environment that help the programmer in constructing programs. Some examples of these tools, currently available, are (syntax-directed) editors, debuggers, compilation aids (like the *make* program on UNIX machines) and so on, but also hierarchical file systems and facilities to communicate with other programmers on the same computer, or even on a network. In designing a new programming language more and more attention is being paid to such an environment. See for example Ada [100], where a set of requirements has been defined that a programming environment should meet to be called an *Ada Program Support Environment* (APSE).

An example of an even more integrated system is the IOTA programming system [92] that offers a set of programs consisting of a syntax-directed editor, compiler, debugger and a correctness prover, that all work on a huge database in which program modules are being represented in some internal

† Ada® is a trademark of the United States Government, Ada Joint Program Office

‡ UNIX™ is a trademark of Bell Laboratories, Incorporated

representation. In this way the cooperation between programmers and the reusability of software modules is highly improved.

1.3. Parallel Programming

With the introduction of parallelism in programming two main approaches can be seen in the field of programming languages. On the one hand, existing programming languages have been extended to incorporate features that deal with parallelism like Concurrent Pascal [32], on the other hand new languages have been developed that are suitable for writing parallel programs from the start, like OCCAM [65] and Ada [15]. We think that with the introduction of concurrency in programming languages, programmers have to start thinking of solving problems in a parallel way, as opposed to the sequential way of thinking imposed by the Von Neumann computers. Therefore we are in favour of programming languages especially designed to support concurrency. Such programming languages will be the first step towards real parallel programming.

1.4. Mathematical Concurrency Theories

ACP (Algebra of Communicating Processes, [24]), or more informal process algebra, is one of the many mathematical theories for concurrency. Other examples from this family of theories are: CSP, CCS, Petri nets, trace theory, temporal logic and denotational semantics. Specifications in ACP have been applied to a large domain of problems ranging from communication protocols [25], [4], algorithms for systolic systems [109] and electronic circuits [13] up to CIM architectures [83]. The manual verification of specifications of small size has already been achieved, but for industrially relevant problems we feel the need for a set of computer tools to help us with the specification, simulation, verification and implementation. These tools will together form a programming or *specification environment*. The first requirement for building such a set of tools is a specification language that is based on ACP.

1.5. Motivation for PSF

A first attempt has been made in [82] to give an algebraic specification of ACP. In this paper it is concluded that the transformation of process algebra into an algebraic specification is quite easy, but that the transformation of an application of process algebra into an algebraic specification takes more effort. It was also stated in this paper, that in specifying process algebra applications in some formal language, one has to be more accurate with respect to the specification of the data types and the ports at which processes communicate. Another disadvantage of specifying process algebra applications by means of algebraic specifications is that the specifications do not have the same appearance as the ones we are used to.

To deal with these problems we have decided to start the development of a new specification formalism called PSF (Process Specification Formalism) that would be based on two concepts. Firstly, the specification of the data types must be performed in some algebraic specification formalism and secondly, the specification of the behaviour of processes must be performed in some formalism especially designed for this need. Another example of such a project is LOTOS [67] in which ACT ONE [40] has been combined with CCS [90]. We have chosen ASF [20] as the algebraic specification formalism and ACP as the base for the formalism as presented in this chapter. For a short explanation of ASF, see section 2.3 and for a short explanation of ACP, see the following section.

2. DEFINITION & DESCRIPTION

2.1. EXPLANATION OF ACP

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

2.1.1. General Setting

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

In ACP, on the contrary, a set of rules is defined first. These rules hold in most models that have been proposed. This way we get a more common algebraic theory, such that whenever a new rule is introduced we can find out in which class of models it holds and in which it does not. We can try to describe a model just by rules and we can find out which operators can be defined in a model and which ones cannot. Because of this approach we are able to compare theories of concurrency and the pros and cons of, say, CSP and CCS can be discussed. See for example [46].

2.1.2. An Introduction

In this section we will give a brief introduction to ACP. This introduction is by no means intended to be complete, but merely gives an intuitive notion of what we are dealing with. For a complete introduction to ACP we refer to [24] and [14].

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

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

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

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

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

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

figure 2.1 Basic Process Algebra.

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

One might think that the axiom $x(y + z) = xy + xz$ is missing. However, this axiom was left out on purpose, because $x(y + z)$ represents something else than $xy + xz$, i.e. we want to consider models of the theory where they are different. The difference originates from the moment of choice. An example, due to Peter Weijland [14], will explain this difference.

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

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

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

2.1.3. Axiomatization of ACP_τ

For completeness, we will list the axiomatization of the theory ACP_τ in the following table. In order to define the merge operator, we need two auxiliary operators. The first one is the left-merge, which is equal to the merge operator but has the constraint that the left argument must start with executing an action. The second one is the communication merge (|), which has the constraint that the first actions of the two arguments must communicate.

$x + y = y + x$	A1	$x\tau = x$	T1
$(x + y) + z = x + (y + z)$	A2	$\tau x = \tau x + x$	T2
$x + x = x$	A3	$a(\tau x + y) = a(\tau x + y) + ax$	T3
$(x + y)z = xz + yz$	A4		
$(xy)z = x(yz)$	A5		
$x + \delta = x$	A6		
$\delta x = \delta$	A7		
$a\|b = b\|a$	C1		
$(a\|b)\|c = a\|(b\|c)$	C2		
$\delta\|a = \delta$	C3		
$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x + x\|y$	CM1		
$a \mathbin{\lfloor\!\lfloor} x = ax$	CM2	$\tau \mathbin{\lfloor\!\lfloor} x = \tau x$	TM1
$ax \mathbin{\lfloor\!\lfloor} y = a(x \parallel y)$	CM3	$\tau x \mathbin{\lfloor\!\lfloor} y = \tau(x \parallel y)$	TM2
$(x + y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$	CM4	$\tau\|x = \delta$	TC1
$ax\|b = (a\|b)x$	CM5	$x\|\tau = \delta$	TC2
$a\|bx = (a\|b)x$	CM6	$\tau x\|y = x\|y$	TC3
$ax\|by = (a\|b)(x \parallel y)$	CM7	$x\|\tau y = x\|y$	TC4
$(x + y)\|z = x\|z + y\|z$	CM8		
$x\|(y+z) = x\|y + x\|z$	CM9	$\partial_H(\tau) = \tau$	DT
		$\tau_I(\tau) = \tau$	TI1
$\partial_H(a) = a$ if $a \notin H$	D1	$\tau_I(a) = a$ if $a \notin I$	TI2
$\partial_H(a) = \delta$ if $a \in H$	D2	$\tau_I(a) = \tau$ if $a \in I$	TI3
$\partial_H(x+y) = \partial_H(x) + \partial_H(y)$	D3	$\tau_I(x+y) = \tau_I(x) + \tau_I(y)$	TI4
$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	D4	$\tau_I(xy) = \tau_I(x) \cdot \tau_I(y)$	TI5

figure 2.2 ACP_τ.

2.2. Explanation of SDF

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

2.2.1. General Aspects

SDF, as introduced in [56], stands for: 'Syntax Definition Formalism'. It is a language to specify the lexical syntax, context-free syntax and abstract syntax of programming languages in a formal way and can be seen as an alternative to LEX [76] and YACC [70]. It is possible to generate a lexical scanner and parser from such an SDF-definition. These parse tables together with a universal parser form a parser for the specified language. It is also possible to generate a so-called syntax directed editor from a description of the layout and the parse tables. This whole system is being implemented in LISP as part of ESPRIT

Project 2177: GIPE II (Generation of Interactive Programming Environments). We will use the SDF language from [56] with the additions and changes as described in [58].

2.2.2. SDF Syntax

An SDF definition contains the description of the *lexical syntax* and of the *context-free syntax* of a language. The notion *sort* corresponds to non-terminals and the *function* declared in the other sections correspond to production rules as used in BNF grammars [1].

Following is an adaptation of an example taken from [20].

```
module example
  exports
    sorts
      Digit Letter Int Id Id-tail Comment-char
    lexical syntax
      [a-z]                   -> Letter
      [0-9]                   -> Digit
      Digit+                  -> Int
      [a-z0-9]                -> Id-tail
      Letter Id-tail*         -> Id
      [ \n\t]                 -> LAYOUT
      ~[{}]                   -> Comment-char
      "{" Comment-char* "}"   -> LAYOUT
    sorts
      Expr
    context-free syntax
      Expr "+" Expr      -> Expr        assoc
      Expr "*" Expr      -> Expr        assoc
      "(" Expr ")"       -> Expr        bracket
      Id                 -> Expr
```

figure 2.3 A small SDF example.

We will point out some of the SDF constructions that appear in this example. The *sorts* section contains the names of the non-terminals of the grammar which can be derived from an SDF-specification. The lexical syntax section specifies part of the regular grammar which is used to generate a lexical analyzer. Elements of the context-free syntax may be interspersed with strings belonging to the predefined sort LAYOUT. The latter will be skipped by the lexical analyzer generated from the SDF definition. The function declaration may be composed of other lexical sorts, (negated) character classes, terminals and list expressions. In the lexical syntax section two kinds of list expressions are allowed:

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

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

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

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

2.3. Explanation of ASF

ASF is an algebraic specification formalism that emerged from the so-called 'PICO-formalism' [19] and which is fully described in [20]. An implementation of ASF is described in [57].

We are using ASF as the basis for the modularization concepts of PSF and for the specification of abstract data types in PSF. Because most aspects of ASF specifications will appear in the description of PSF we will not discuss them here. For specific information we refer to [20].

2.4. An Informal Description of PSF

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

2.4.1. Modules

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

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

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

figure 2.4 The different sections in modules.

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

2.4.2. Lexical Syntax

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

- Layout:

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

- Comments:

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

- Identifiers:

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

- Keywords:

  The following identifiers are reserved keywords:

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

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

- Operators:

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

Layout characters and comments may separate identifiers in PSF but may never occur embedded in a lexical token. In cases of ambiguity, the longest token is preferred. For detailed information on the lexical syntax of ASF we refer to [20].

2.4.3. Modularization

There are some constructs in PSF that help to make specifications in a modular fashion. The three sections that deal with this feature are the export, import and parameter section. Along with a short description of each modularization concept we will give the associated *structure diagrams*, as introduced in [20].

Module *A* is represented by a rectangular box.

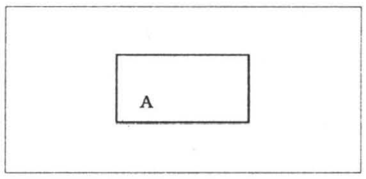


figure 2.5 Structure diagram of a module.

2.4.3.1. Export

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

2.4.3.2. Import

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

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

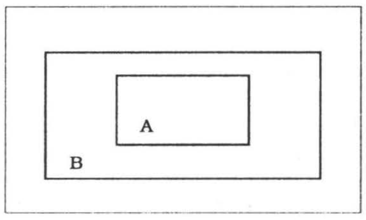


figure 2.6 Structure diagram of an import.

2.4.3.3. Parameters

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

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

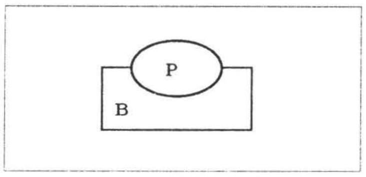


figure 2.7 Structure diagram of module with a parameter.

2.4.4. Module Expressions

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

2.4.4.1. Renaming

The visible names of a module can be renamed by use of the *renamed by* construct, which specifies a renaming by giving a list of pairs of renamings in the form of an old visible name and a new visible name. It is not possible to rename just one of the instances of an overloaded name. So if a renaming is applied to an overloaded name, all instances of this name will be renamed.

2.4.4.2. Parameter Binding

The *bound by* construct is used to bind parameters and specifies the name of a parameterized module, a parameter name, a list of bindings (pairs consisting of a formal name and an actual name), and the name of an actual module. As a result of parameter binding, a parameter is replaced by a name from the actual module as specified by the list of bindings. Therefore a parameter can only be bound once. Parameter binding should obey the following rules:

- The actual names must be visible outside the actual module.
- Formal names and actual names must be of the same kind, i.e., they both should be atoms, sorts etc. Furthermore their input type and output type should be the same.
- All names in a parameter should be bound to a name of the actual module.

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

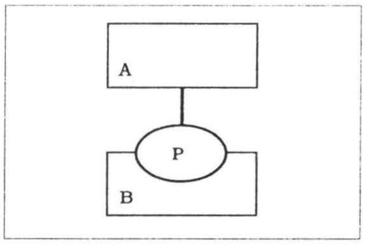


figure 2.8 Structure diagram of parameter binding.

2.4.5. Data Specification

There are some sections that are specific for the specification of data types. These sections are explained below. For more specific information about the specification of the data types we refer to [20].

2.4.5.1. Sorts & Functions, Signatures

As we have seen, the declaration of sorts and functions can occur in two places. Declarations can occur in the export section of a module, so that they are visible, or they can be declared as being *hidden*. In the *sorts* section we define which sorts are introduced. The functions are declared within the *functions* section along with their *input type* (the type of the arguments) and their *output type*. The combination of an input type and an output type is simply called the *type* of a function. Functions without arguments will be called *constants*. Declarations of sorts and functions over these sorts are called *signatures*. See, for instance, [40] for a description of the notion of signatures.

2.4.5.2. Equations

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

An (unconditional) equation has the following form:

[*tag*] $t_l = t_r$

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

Conditional equations can have two (equivalent) forms:

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

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

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

Variables in equations are implicitly universally quantified.

2.4.6. Process Specification

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

2.4.6.1. Sets

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

set-name = *set-expression*

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

- Just the name of a *sort* denotes the set of all well-formed terms that are typed with the specified sort. Sorts do not have to be declared as sets.
- We are able to construct sets by enumerating terms: { $t_1, t_2, \ldots, t_n$ }. The empty set is denoted by {}.
- Because it is impossible to enumerate infinite sets we need some weak form of *replacement* in which the variables can only range over the domain of a given set. So we introduce the so-called *placeholder* construction which is used in the next example to define a set A that consists of the terms that can be obtained by applying a certain function f to all elements of S: In this example t is the variable, which we will call a *placeholder* in the sequel.

  A = { $f(t)$ | t in S }

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

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

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

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

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

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

2.4.6.2. Atomic Actions

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

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

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

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

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

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

figure 2.9 Example of the use of the predefined sort *atoms*.

2.4.6.3. Communication

Communication in PSF can occur between atomic actions only. In such a communication exactly three atomic actions are involved. Two atomic actions which communicate and one that is the resulting *communication action*. We have *handshaking* communication, which means that such a resulting communication action can not participate in any further communication. In the *communications* section we define which atoms can communicate and what the result of this communication will be.

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

$a \mid b = c$

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

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

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

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

A communication definition should be given for all atomic actions that are visible within a module. Whenever a communication is not listed in the *communications* section, it is thought of as being a communication resulting in deadlock (see [24]).

2.4.6.4. Variables

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

2.4.6.5. Processes

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

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

P_1
P_2 : $s_1 \# s_2$

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

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

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

Process expressions are defined by means of induction:

- Each atomic action is a process-expression.

- There is one predefined process-expression called *skip* that represents the pre-abstraction from ACP. This feature was introduced in [5] where the atomic action t is used.
- There are three binary operators on process-expressions:
  - sequential composition : $PE_1 \,.\, PE_2$
  - alternative composition or choice : $PE_1 + PE_2$
  - parallel composition or merge : $PE_1 \mid\mid PE_2$

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

  Note that we will not need the ACP constant δ or the (auxiliary) operators $\mid$, $\Vert\!\underline{}$ (as in [14]).

2.4.6.6. Scope of Placeholders

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

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

2.5. Definition of PSF in SDF

2.5.1. Why Use SDF?

In this section we will give the definition of the PSF formalism in SDF. The reason that we have chosen to use SDF instead of, for instance, a BNF grammar is that we found the former to be much more formal and that we had the possibility to check the PSF grammar during the development by means of the error messages generated by the parser generator described in [20]. Moreover, we could easily build a prototype parser that was able to check our PSF specifications for any grammatical inconsistencies.

2.5.2. Definition of PSF

This definition of PSF does not contain the syntax of the data modules. We refer to [20] for a description of the syntax of the algebraic specifications in ASF. Nevertheless, the syntax for the modularization concepts, which is borrowed from ASF, is included. Note that the constraint that comments must follow a layout character is not expressed in the following definition.

```
module PSF
exports
  sorts
    Id-char Id-body Ident
    Op-symbol Operator
    Com-char Com-end

  lexical syntax
    [0-9a-zA-Z']                               -> Id-char
    [0-9a-zA-Z'\-]                             -> Id-body
    Id-char                                    -> Ident
    Id-char Id-body* Id-char                   -> Ident

    [!@$%\^&+\-*;?~/|\\]                       -> Op-symbol
    Op-symbol+                                 -> Operator
    "." Ident "."                              -> Operator

    [ \n\t]                                    -> LAYOUT

    ~ [\n\-]                                   -> Com-char
    "-" ~ [\n\-]                               -> Com-char
    "--"                                       -> Com-end
    "\n"                                       -> Com-end
    "-\n"                                      -> Com-end
    "--" Com-char* Com-end                     -> LAYOUT

  sorts
    Specification Process-module Parameters Parameter Exports
    Imports Module-expression Modifier Renamed
    Renamings Renaming Binding Bound Psf-sorts
    Psf-functions Psf-function Atoms Atom-declaration Processes
    Process-decl Sets Set-defs Set-definition Sets-param
    Set-param-defs Set-exp Placeholder
    Set-item Variables Vars
    Communications Communication Communication-def
  Atom-exp Definitions
  Process-def Process-def-head
  Process-exp Predef-process Set-operator
  Identifier Ident-or-op
  Term Primary

context-free syntax
  Ident                                          -> Identifier
  Process-module+                                -> Specification
  "process" "module"  Identifier
     "begin"
        Parameters
        Exports
        Imports
        Atoms
        Processes
        Sets
        Communications
        Variables
        Definitions
     "end" Identifier                            -> Process-module

  "parameters" { Parameter "," }+                -> Parameters
                                                 -> Parameters
  Identifier
  "begin"
     Psf-sorts
     Psf-functions
     Atoms
     Processes
     Sets-param
  "end" Identifier                               -> Parameter

  "exports"
    "begin"
        Atoms
        Processes
        Sets
     "end"                                       -> Exports
                                                 -> Exports

                                                 -> Imports
  "imports" {Module-expression ","}+             -> Imports
  Identifier                                     -> Module-expression
  Identifier "{" Modifier "}"                    -> Module-expression
  Renamed                                        -> Modifier
  Bound                                          -> Modifier
  Renamed Bound                                  -> Modifier
  Bound Renamed                                  -> Modifier

  "renamed" "by" Renamings                       -> Renamed
  "[" {Renaming ","}+ "]"                        -> Renamings
  Ident-or-op "->" Ident-or-op                   -> Renaming

  Identifier                                     -> Ident-or-op
  "_" Operator "_"                               -> Ident-or-op
  Operator "_"                                   -> Ident-or-op

  Binding+                                       -> Bound
```

```
Identifier "bound" "by" Renamings "to" Identifier
                                             -> Binding

"sorts" {Identifier ","}+                    -> Psf-sorts
                                             -> Psf-sorts

"functions" Psf-function+                    -> Psf-functions
                                             -> Psf-functions
Identifier ":" {Identifier "#"}* "->" {Identifier "#"}+
                                             -> Psf-function
Operator "_" ":" Identifier "->" Identifier  -> Psf-function
"_" Operator "_" ":" Identifier "#" Identifier
                     "->" {Identifier "#"}+  -> Psf-function

"atoms" Atom-declaration+                    -> Atoms
                                             -> Atoms
{Identifier ","}+                            -> Atom-declaration
{Identifier ","}+ ":" {Identifier "#"}+      -> Atom-declaration

"processes" Process-decl+                    -> Processes
                                             -> Processes
{Identifier ","}+                            -> Process-decl
{Identifier ","}+ ":" {Identifier "#"}+      -> Process-decl

"sets" Set-defs+                             -> Sets
                                             -> Sets

"of" Identifier Set-definition+              -> Set-defs
"of" "atoms" Set-definition+                 -> Set-defs
Identifier "=" Set-exp                       -> Set-definition

"sets" Set-param-defs+                       -> Sets-param
                                             -> Sets-param
"of" Identifier Identifier+                  -> Set-param-defs
"of" "atoms" Identifier+                     -> Set-param-defs

Identifier                                   -> Set-exp
"{" { Set-item "," }* "}"                    -> Set-exp
"(" Set-exp ")"                              -> Set-exp bracket
Set-exp "+" Set-exp                          -> Set-exp assoc
Set-exp "." Set-exp                          -> Set-exp assoc
Set-exp "\\" Set-exp                         -> Set-exp
"{" {Set-item ","}+ "|" {Placeholder ","}+ "}"
                                             -> Set-exp

Identifier                                   -> Set-item
Identifier "(" { Term "," }+ ")"             -> Set-item

Identifier "in" Identifier                   -> Placeholder

"variables" Vars+                            -> Variables
                                             -> Variables
{ Identifier "," }+ ":" "->" Identifier      -> Vars

"communications" Communication-def+          -> Communications
                                             -> Communications
Communication                                -> Communication-def
Communication "for" {Placeholder ","}+       -> Communication-def
Atom-exp "|" Atom-exp "=" Atom-exp           -> Communication
```

```
Identifier                                        -> Atom-exp
Identifier "(" { Term "," }+ ")"                  -> Atom-exp

"definitions" Process-def+                        -> Definitions
                                                  -> Definitions
Process-def-head "=" Process-exp                  -> Process-def
Identifier                                        -> Process-def-head
Identifier "("  { Term "," }+ ")"                 -> Process-def-head

Predef-process                                    -> Process-exp
Process-def-head                                  -> Process-exp
"(" Process-exp ")"                               -> Process-exp bracket
Process-exp "." Process-exp                       -> Process-exp assoc
Process-exp "+" Process-exp                       -> Process-exp assoc
Process-exp "||" Process-exp                      -> Process-exp assoc

Primary                                           -> Term
Term Operator Primary                             -> Term
Identifier                                        -> Primary
Identifier "(" {Term ","}+ ")"                    -> Primary
Operator Primary                                  -> Primary
"(" Term ")"                                      -> Primary

"sum" "(" Placeholder "," Process-exp ")"         -> Process-exp
"merge" "(" Placeholder "," Process-exp ")"       -> Process-exp
Set-operator "(" Identifier "," Process-exp ")"
                                                  -> Process-exp

"skip"                                            -> Predef-process
"encaps"                                          -> Set-operator
"hide"                                            -> Set-operator
```

figure 2.10 Specification of PSF.

2.6. Semantical Constraints

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

2.6.1. Overloading

It is allowed to *overload* the names of functions, atoms and processes. This means that the same name can be used to denote different functions, atoms or processes. An example for a function *f*:

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

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

Overloaded functions, atoms and processes should have unique input types. This restriction forbids overloaded constants and multiple declaration of names with the same type. Thus the following is not allowed.

```
f:    s1 -> s2
f:    s1 -> s3        --not allowed
```

Variables and sets cannot be overloaded. This restriction forbids multiple declaration of variable names and set names within one module.

2.6.2. Typing of Terms

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

2.6.3. Communication

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

2.6.3.1. Firm Handshaking

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

```
atoms
   r,s,c : NATURAL
communications
   r(0) | s(0) = c(0)
   c(1) | s(1) = r(1)
```

figure 2.11 A violation of firm handshaking.

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

2.6.3.2. Consistency of Export

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

2.6.3.3. Consistency of Communications

The definition, in separate modules, of the result of a communication may lead to inconsistencies when putting these modules together. We call two modules inconsistent with respect to their communications whenever there exists a communication between two atomic actions that is defined in both modules and yields two different atomic actions. It is not allowed to combine two inconsistent modules. To be more precise, the following situations are not allowed:

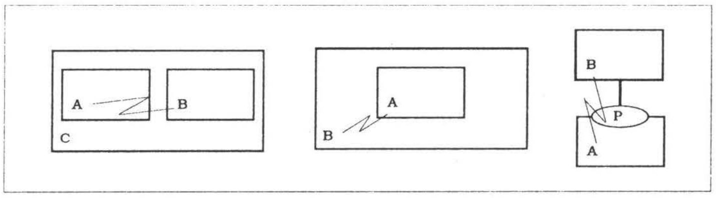


figure 2.12 Three ways of illegally combining inconsistent modules.

Whenever two modules are inconsistent they may not be

- imported into a third module.
- imported into each other.
- bound to each other's parameters.

In figure 2.13 two examples of such an inconsistency are shown: In the first example modules A and B are inconsistent with respect to their communications, because A defines $r \mid s$ to be c and B defines $r \mid s$ to be d. In determining whether two modules are consistent we should not only consider the explicitly defined communications, but also the assumption that all communications, between atomic actions that are visible in a module, that have not been defined in the *communications* section, are implicitly defined to be deadlock. The second example in figure 2.13 illustrates this situation. Module P does not list a communication between a and d, and so this communication is defined to yield deadlock. However, module Q tries to re-define this communication by $a \mid d = e$, which is illegal.

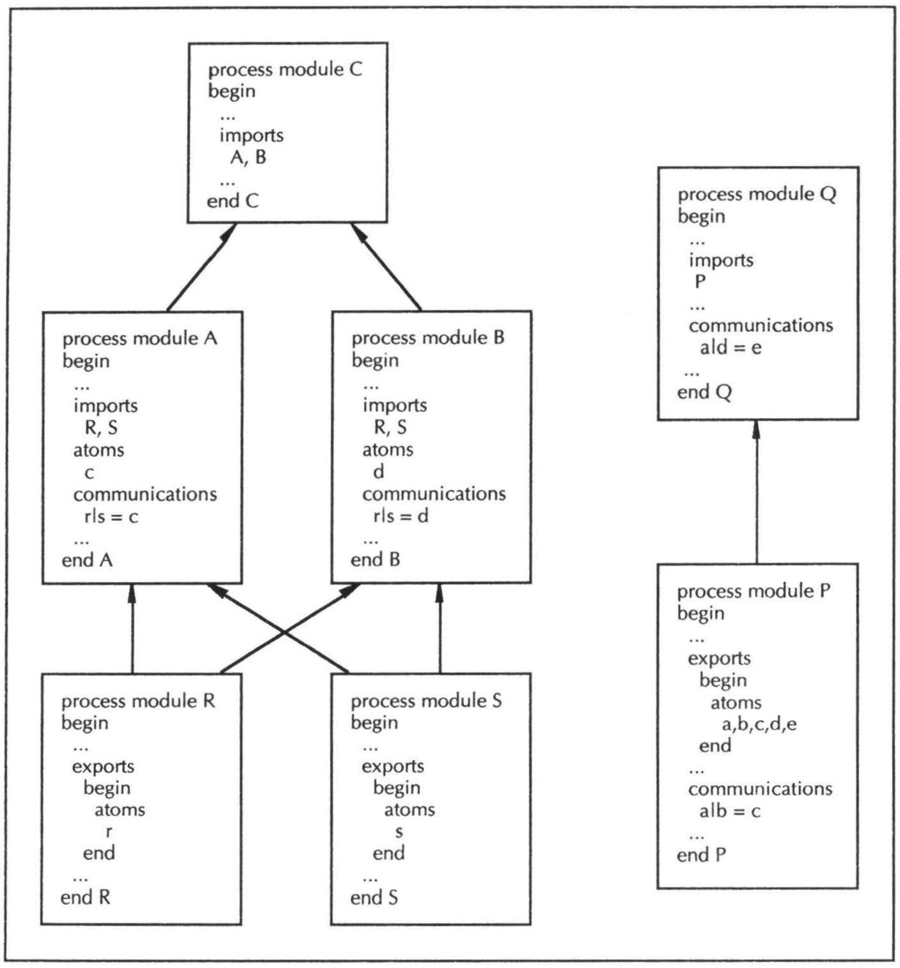


figure 2.13 Two possible sources of inconsistency.

2.6.4. Variables

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

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

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

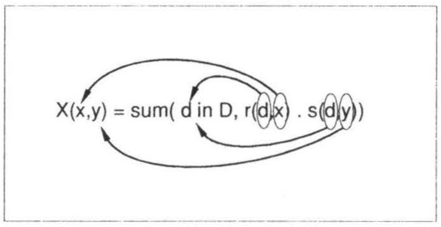


figure 2.14 Binding of variables in a process expression.

3. SEMANTICS

3.1. SEMANTICS FOR PSF

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

First, we have to define the semantics for the data specification part. It is quite natural to choose the same semantics as the one chosen for ASF, i.e. the *initial algebra semantics* as in [40] and [48]. Ideally, the set of equations takes the form of a complete term rewriting system [71], so that equality of terms can be determined by reducing to *normal form*, and the set of normal forms is isomorphic to the initial algebra. For the section that defines the processes it is convenient to use another kind of semantics because, by nature, it has more in common with *transition networks*. (Note, on the other hand, that in [82] we find an attempt to give an algebraic specification of ACP using initial algebra semantics.) For this process section we use an *operational semantics* that is defined with the aid of *action relations* [95], which will be presented in section 3.7.3. Action relations for ACP have been introduced in [47]. On top of these action relations we can define a semantics, such as *bisimulation semantics* or *failure semantics*. Finally we also give a semantics for the sets and the atomic actions, which will both be derived from the initial algebra semantics. The dependencies among the semantics of the different parts of PSF are expressed by the following picture:

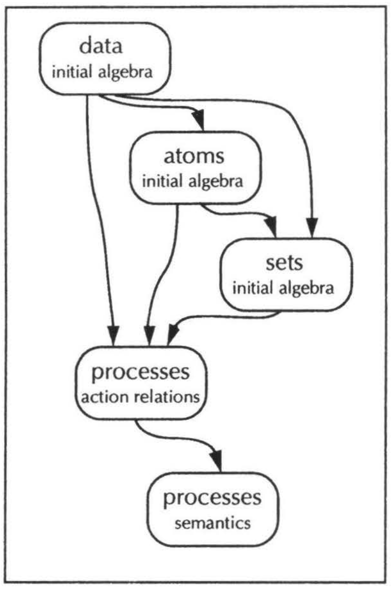


figure 3.1 Dependencies among different semantical domains.

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

3.2. The Origin Rule

Because a PSF specification may consist of several modules, there may arise some problems with multiple declarations of the same name when putting these modules together in case of import. We don't want any unintended and unexpected name identifications, so we introduce the so-called *origin rule*, in a similar way as is done in ASF [20] to locate the defining position of each occurrence of an identifier. In contrast with ASF, the origin rule in PSF is also defined for parameters. This extension is due to Hans Mulder.

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

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

- s: The section of the module in which the declaration of *n* occurs:
  - s = <*p-name*> for a parameter section with name *p-name*,
  - s = *par* if the object is a parameter,
  - s = *export* for the export section,
  - s = *hidden* for the sort, function, atom, process, set and variable sections outside the export section.
- c: The subcategory to which *n* belongs:
  - c = *par* for a parameter,
  - c = *sort* for a sort name,
  - c = *function* for a function name,
  - c = *atom* for an atom name,
  - c = *process* for a process name,
  - c = *set* for a set name,
  - c = *variable* for a a variable name.
- n: The name as introduced by the declaration. This name is extended by the input type for functions, atoms and processes. For parameters it is extended by the signature of the objects belonging to the parameter. (This signature consists of a collection of names plus origins.)

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

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

The origin rule:

- Two visible sorts, functions, atoms, processes or sets are identical if they both have the same name and the same origin. For functions we also require that they have the same output sort and for sets we require that they are of the same sort. Visible sorts, functions, atoms, processes and sets having the same name but different origin are forbidden. Functions with the same name and the same origin, but with different output type are forbidden. Sets with the same name and the same origin, but of a different type are also forbidden.
- Two hidden sorts or hidden sets are identical if they have the same origin.

- Two variables are identical if they have the same origin and if the corresponding types (sorts) can be identified using the aforementioned rules.
- Two hidden functions, atoms and processes are identical if they have the same origin and if the two corresponding types have equal structure and can be identified componentwise using the first two rules given above.
- Two parameters are identical if they both have the same name and the same origin. Parameters having the same name but different origin are forbidden.
- A set and a sort may not have identical names (possibly after the implicit renaming for imported hidden objects). The same holds for a variable together with a function without input. In the same way, an atomic action and a process may not have the same name if they have identical input type.

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

3.3. NORMALIZATION

In order to be able to assign a semantics to a PSF specification we have to assign a semantics to each module. The semantics of a module can only be determined in its context, being the total specification. This evaluation of a module in its context leaves us with a so-called *normal form*.

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

How this normalization should be performed, is described in [103].

3.4. SEMANTICS FOR DATA TYPES

As the semantics for the data types we use the initial algebra semantics as defined in [40] and [48]. We assume that all modularization concepts from the data modules have been removed by the normalization procedure, thus leaving a *flat* algebraic specification. To define this initial algebra we first need to introduce some other notions.

- A *signature* Σ is a collection of names of sorts and functions. To each function name we associate a list of sort names that represent the input type and one sort name for the output type. Functions with an empty input type are constants of the specified output type.

- The set V consists of variables. To each variable a sort is associated.
- A *term* is a construction of functions and variables with correctly typed arguments, defined inductively:
  - Each variable associated with sort S, is a term of sort S.
  - If $t_1 \dots t_n$ are terms of sort $S_1 \dots S_n$, and function f has input type $S_1 \times \dots \times S_n$ and output type T, then $f(t_1, \dots, t_n)$ is a term of sort T.

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

  $$[f(t_1, \dots, t_n)] = [f]\,([t_1], \dots, [t_n])$$
- An equation of two terms is true in a Σ-algebra A, whenever the interpretation of both terms denotes the same element.

  $$A \models t_1 = t_2 \iff [t_1] =_A [t_2]$$

  If all equations, $t_1 = t_2$, in E, are valid in the Σ-algebra A, we write $A \models E$.
- The class of Σ-algebras A with $A \models E$. is denoted by $Alg(\Sigma,E)$. This class contains one special algebra called the *initial algebra* of (Σ,E), $I(\Sigma,E)$.

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

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

3.5. Set Semantics

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

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

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

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

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

  $[s_1 \setminus s_2] = [s_1] \setminus [s_2]$ for sets s_1 and s_2 of sort S;
- $[\{t_1(\underline{u}), \dots, t_n(\underline{u}) \mid u_1 \in D_1, \dots, u_m \in D_m\}] =$

  $[\{t_1(\underline{u}) \mid u_1 \in D_1, \dots, u_m \in D_m\}] \cup \dots \cup [\{t_n(\underline{u}) \mid u_1 \in D_1, \dots, u_m \in D_m\}]$;
- $[\{t(\underline{u}) \mid u_1 \in D_1, \dots, u_m \in D_m\}] = \{[t(\underline{u})] \mid u_1 \in \text{Repr}(D_1), \dots, u_m \in \text{Repr}(D_m)\}$.

3.6. Semantics for Atomic Actions

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

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

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

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

3.7. Operational Semantics

3.7.1. Action Rules

In this section we will define the operational semantics for the process definition part of PSF with the aid of so-called *action rules*. These action rules have been used already in other concurrency theories, see for example [95] in which Plotkin gives the operational semantics for CSP [61]. Action rules in ACP are introduced in [47]. But first we will have a look at what a process definition stands for.

3.7.2. Process Definitions

A process definition in general looks like:

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

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

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

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

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

figure 3.2 Three different representations of the same process expression.

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

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

```
processes
    X, Y: BOOLEAN # NATURAL

variables
    b : -> BOOLEAN
    n : -> NATURAL

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

figure 3.3 An abbreviation of process definitions.

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

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

figure 3.4 Part of the expanded *definitions* section.

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

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

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

3.7.3. Action Rules for PSF

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

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

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

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

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

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

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

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

- $R.||$
  - $a \mid b = c$ means that the communication between a and b has been defined to be c.
- *R.encaps*:
  - H : the set of atomic actions that have to be encapsulated.
- *R.hide*:
  - I : the set of atomic actions that have to be renamed into *skip*.
- *R.rec*:
  - $\underline{u} \in \underline{D}$ means $u_1 \in D_1, u_2 \in D_2, \ldots, u_n \in D_n$
    - $\underline{u} = (u_1, u_2, \ldots, u_n)$
    - $\underline{D} = (D_1, D_2, \ldots, D_n)$
    - D_i is a sort.
  - $y(\underline{u})$: a process expression with a list of terms $\underline{u} \in \underline{D}$ as parameters.
  - X : a process name declared as $X : D_1 \# D_2 \# \ldots \# D_n$
  - $X(\underline{u}) = y(\underline{u})$: an equation from the *definitions* section.
- *R.sum*:
  - D : a set.
  - $u \in D$: u is an element of *repr(D)*.
  - d in D: d is a variable over the sort associated with the set D.
- *R.merge*:
  - $|D|$: the number of elements in set D.

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

R.merge1
$$\frac{|D|>1;\ u\in D;\ x(u) \xrightarrow{a} x'}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{a} \mathbf{merge}(d\ \mathbf{in}\ D\backslash\{u\},\ x(d))\ ||\ x'}$$

R.merge2
$$\frac{|D|>1;\ u\in D;\ x(u) \xrightarrow{a} \surd}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{a} \mathbf{merge}(d\ \mathbf{in}\ D\backslash\{u\},\ x(d))}$$

R.merge3
$$\frac{|D|=1;\ u\in D;\ x(u) \xrightarrow{a} x'}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{a} x'}$$

R.merge4
$$\frac{|D|=1;\ u\in D;\ x(u) \xrightarrow{a} \surd}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{a} \surd}$$

R.merge5
$$\frac{|D|>2;\ u\in D;\ v\in D;\ u\neq v;\ x(u) \xrightarrow{a} y;\ x(v) \xrightarrow{b} z;\ a|b=c}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{c} (\mathbf{merge}(d\ \mathbf{in}\ D\backslash\{u,v\},\ x(d))\ ||\ y)\ ||\ z}$$

R.merge6
$$\frac{|D|>2;\ u\in D;\ v\in D;\ u\neq v;\ x(u) \xrightarrow{a} y;\ x(v) \xrightarrow{b} \surd;\ a|b=c}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{c} \mathbf{merge}(d\ \mathbf{in}\ D\backslash\{u,v\},\ x(d))\ ||\ y}$$

R.merge7
$$\frac{|D|>2;\ u\in D;\ v\in D;\ u\neq v;\ x(u) \xrightarrow{a} \surd;\ x(v) \xrightarrow{b} \surd;\ a|b=c}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{c} \mathbf{merge}(d\ \mathbf{in}\ D\backslash\{u,v\},\ x(d))}$$

R.merge8
$$\frac{|D|\leq 2;\ u\in D;\ v\in D;\ u\neq v;\ x(u) \xrightarrow{a} y;\ x(v) \xrightarrow{b} z;\ a|b=c}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{c} y\ ||\ z}$$

R.merge9
$$\frac{|D|\leq 2;\ u\in D;\ v\in D;\ u\neq v;\ x(u) \xrightarrow{a} y;\ x(v) \xrightarrow{b} \surd;\ a|b=c}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{c} y}$$

R.merge10
$$\frac{|D|\leq 2;\ u\in D;\ v\in D;\ u\neq v;\ x(u) \xrightarrow{a} \surd;\ x(v) \xrightarrow{b} \surd;\ a|b=c}{\mathbf{merge}(d\ \mathbf{in}\ D,\ x(d)) \xrightarrow{c} \surd}$$

figure 3.6 Table of action relations.

3.7.4. Process Semantics

Now that we have defined the action relations for PSF we are able to assign a semantics to processes. In this case we define *bisimulation* [94] on top of these action relations.

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

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

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

$\underline{\leftrightarrow}$ is a *congruence* on process expressions. See [12] for a proof.

3.7.5. An Example

Suppose we have the following definition of a certain process X: N is the sort of the naturals and B is the sort of the booleans.

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

figure 3.7 Definition of process *X*.

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

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

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

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

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

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

$$\textbf{R.rec1:}\ \ \frac{\begin{array}{c} X = \textbf{sum}(d\ \textbf{in}\ N,\ \textbf{sum}(e\ \textbf{in}\ B,\ r(d,e)\cdot Y(d,not(e))));\\ \textbf{sum}(d\ \textbf{in}\ N,\ \textbf{sum}(e\ \textbf{in}\ B,\ r(d,e)\cdot Y(d,not(e)))) \xrightarrow{r(5,false)} Y(5,not(false))\end{array}}{X \xrightarrow{r(5,false)} Y(5,not(false))}$$

figure 3.8 The derivation of a transition.

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

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

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

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

$$Y(5,not(false)) = s(5)\cdot s(not(false))\cdot X;$$

$$\mathbf{R.rec1:}\quad \frac{s(5)\cdot s(not(false))\cdot X \xrightarrow{s(5)} s(not(false))\cdot X}{Y(5,not(false)) \xrightarrow{s(5)} s(not(false))\cdot X}$$

figure 3.9 The derivation of a transition.

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

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

figure 3.10 The derivation of a transition.

3.8. OTHER PROCESS SEMANTICS

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

We assign a graph to each process expression.

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

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

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

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

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

Now that we have given a way of constructing graphs, it is possible to define a wide variety of semantics on this graph domain. These semantics include for example: *trace semantics, failure semantics* ([26], [10]). We can also define *strong observational congruence* [90] on this graph domain, which is in fact equal to our bisimulation semantics as defined in section 3.7.3.

4. EXAMPLES

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

4.1. A LANDING CONTROL SYSTEM

4.1.1. The Problem

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

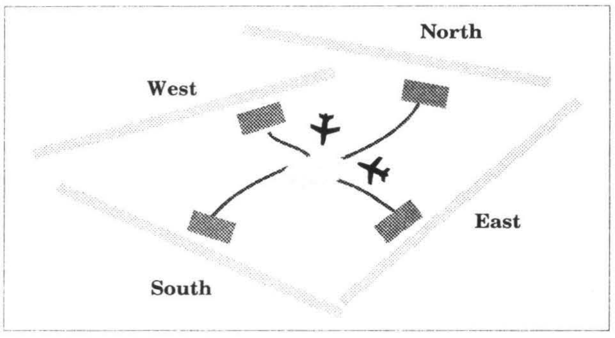


figure 4.1 Timbuktu Airport.

4.1.2. The Implementation

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

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

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

4.1.3. The Specification

```
process module Landing-Control
begin

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

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

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

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

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

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

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

  end Landing-Control
```

figure 4.2 Specification of a generic landing control system.

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

```
data module Timbuktu-Airport
begin

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

end Timbuktu-Airport

process module Timbuktu-Landing-Control
begin

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

end Timbuktu-Landing-Control
```

figure 4.3 Timbuktu Airport definition.

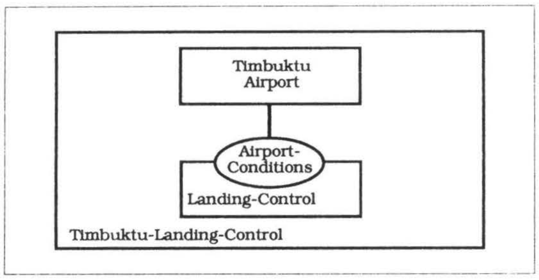


figure 4.4 Timbuktu Airport structure diagram.

4.2. ALTERNATING BIT PROTOCOL

4.2.1. The Problem

One of the most famous communication protocols is the Alternating Bit Protocol (ABP). It has been used many times to serve as a test case for a new formalism. Our specification emanates from the ABP specification in ACP as described in [25].

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

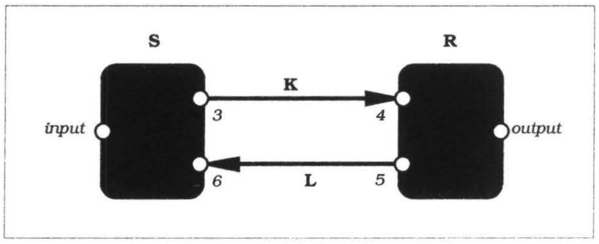


figure 4.5 Graphical representation of the Alternating Bit Protocol.

It consists of four components:

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

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

4.2.1.1. The Sender

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

4.2.1.2. Communication Channel *K*

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

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

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

4.2.1.3. The Receiver

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

4.2.1.4. Communication Channel *L*

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

4.2.2. The Specification

```
data module Booleans
begin

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

```
   variables
     x : -> BOOLEAN

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

end Booleans


data module ABP-Ports
begin

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

end ABP-Ports


data module Errors
begin

  exports
    begin
      sorts
        ERROR
      functions
        ce : -> ERROR
    end

end Errors


data module Bits
begin

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

end Bits
```

```
process module Producer
begin

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

  exports
    begin
      processes
        PROD
    end

  imports
    Bits

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

end Producer


process module Consumer
begin

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

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

  exports
    begin
      processes
        CONS
    end

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

end Consumer
```

```
process module ABP
begin

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

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

  exports
    begin
      processes
        ABP
    end

  imports
    Booleans, ABP-Ports, Errors

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

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

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

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

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

  variables
    b : -> BOOLEAN
    d : -> DATA
```

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

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

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

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

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

end ABP


process module Communication-Ports
begin

  parameters
    Data-Items
      begin
        sorts
          DATA
      end Data-Items

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

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

end Communication-Ports
```

```
process module System-Ports
begin

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

end System-Ports


process module System
begin

  exports
    begin
      processes
       SYS
    end

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

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

end System
```

figure 4.6 PSF specification of the Alternating Bit Protocol.

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

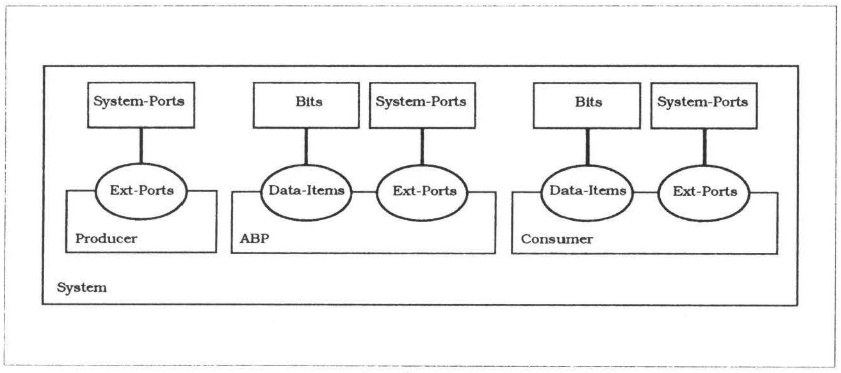


figure 4.7 Structure diagram of the ABP specification.

4.3. A Palindrome Recognizer

4.3.1. The Problem

The algorithm for a palindrome recognizer was introduced in [74]. In [59] the first proof of correctness is given and in [109] we find the ACP version of this proof. The algorithm stems from a class of algorithms for systolic systems. Systolic systems are systems that are constructed from a large number of small cells so that the behaviour of the whole system resembles the behaviour of an individual cell. In this case we will define the behaviour of a cell that will be able to tell whether a string of length two, or less, is a palindrome. Then we construct a true palindrome recognizer by putting many of these cells together in the form of a long chain. A typical cell is shown in figure 4.8.

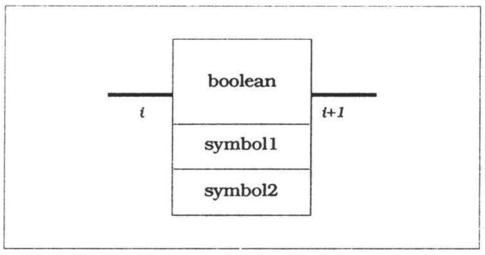


figure 4.8 One cell of the palindrome recognizer.

The ith cell has two communication ports, *i* and *i+1*. It has two locations to store a symbol and one location to store a boolean. A cell can be in one of three states:

1. This is the initial state in which the cell contains no symbols. It represents the empty word in this state and because the empty word is a palindrome it can always write the boolean *true* at port *i*. If a symbol is read at port *i* it is stored in *symbol2*. Then, the boolean value *true* is written to channel *i* because a word consisting of just one symbol is always a palindrome. After this, the cell is in state 2.
2. In this state, another symbol is read from *i* and a boolean from *i+1*, in arbitrary order, and stored in *symbol1* and *boolean*, respectively. Next, the cell is in state 3 .
3. In state 3 the cell contains two symbols and it computes whether the two symbols form a palindrome (i.e. are equal). The result of the calculation : *((symbol1 = symbol2) and boolean)* is written at port *i* and the symbol in *symbol1* is written at port *i+1* leaving room for a new symbol to be read from *i*. The cell is now in state 2 once more.

4.3.2. The Specification

Now we want to specify this palindrome recognizer in PSF. Though this example may seem rather complex, there are only two *process modules* in it. The rest of the specification deals with the data types. The two process modules define the palindrome recognizer from a different point of view. The first process module defines the external behaviour of the process and here we need a rather complicated predicate *is-pal* to determine whether a string is a palindrome. The second specification defines the nature of the cells from which the recognizer is constructed.

```
data module Booleans
begin

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

  variables
    x : -> BOOLEAN

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

```
data module Naturals
begin

  exports
    begin
      sorts
        NATURAL
      functions
        zero  :                     -> NATURAL
        S     : NATURAL             -> NATURAL
        _+_   : NATURAL # NATURAL -> NATURAL
        _*_   : NATURAL # NATURAL -> NATURAL
        equal : NATURAL # NATURAL -> BOOLEAN
    end

  imports
    Booleans

  variables
    x, y : -> NATURAL

  equations
  [P1]   x + zero = x
  [P2]   x + S(y) = S(x+y)
  [M1]   x * zero = zero
  [M2]   x * S(y) = (x*y) + x
  [N1]   equal(zero,zero) = true
  [N2]   equal(zero, S(x)) = false
  [N3]   equal( S(x), zero ) = false
  [N4]   equal( S(x), S(y) ) = equal(x,y)

end Naturals


data module Symbols
begin

    exports
      begin
        sorts
          SYMBOL
        functions
          'a : -> SYMBOL
          'b : -> SYMBOL
          'c : -> SYMBOL
          'd : -> SYMBOL
          'e : -> SYMBOL
          'f : -> SYMBOL
          'g : -> SYMBOL
          'h : -> SYMBOL
          'i : -> SYMBOL
          'j : -> SYMBOL
          'k : -> SYMBOL
          'l : -> SYMBOL
          'm : -> SYMBOL
          'n : -> SYMBOL
          'o : -> SYMBOL
          'p : -> SYMBOL
          'q : -> SYMBOL
          'r : -> SYMBOL
          's : -> SYMBOL
          't : -> SYMBOL
          'u : -> SYMBOL
          'v : -> SYMBOL
          'w : -> SYMBOL
          'x : -> SYMBOL
          'y : -> SYMBOL
          'z : -> SYMBOL
          equal : SYMBOL # SYMBOL -> BOOLEAN
      end

   imports
     Booleans, Naturals

   functions
     ord : SYMBOL -> NATURAL

   variables
     s1,s2 : -> SYMBOL

   equations
   [S1]     ord('a) = zero
   [S2]     ord('b) = S(ord('a))
   [S3]     ord('c) = S(ord('b))
   [S4]     ord('d) = S(ord('c))
   [S5]     ord('e) = S(ord('d))
   [S6]     ord('f) = S(ord('e))
   [S7]     ord('g) = S(ord('f))
   [S8]     ord('h) = S(ord('g))
   [S9]     ord('i) = S(ord('h))
   [S10]    ord('j) = S(ord('i))
   [S11]    ord('k) = S(ord('j))
   [S12]    ord('l) = S(ord('k))
   [S13]    ord('m) = S(ord('l))
   [S14]    ord('n) = S(ord('m))
   [S15]    ord('o) = S(ord('n))
   [S16]    ord('p) = S(ord('o))
   [S17]    ord('q) = S(ord('p))
   [S18]    ord('r) = S(ord('q))
   [S19]    ord('s) = S(ord('r))
   [S20]    ord('t) = S(ord('s))
   [S21]    ord('u) = S(ord('t))
   [S22]    ord('v) = S(ord('u))
   [S23]    ord('w) = S(ord('v))
   [S24]    ord('x) = S(ord('w))
   [S25]    ord('y) = S(ord('x))
   [S26]    ord('z) = S(ord('y))
   [E1]     equal(s1,s2) = equal( ord(s1), ord(s2) )

end Symbols
```

```
data module Strings
begin

  exports
   begin
     sorts
      STRING
     functions
      empty    :                 -> STRING
      _~_      : SYMBOL # STRING -> STRING
      equal    : STRING # STRING -> BOOLEAN
      length   : STRING          -> NATURAL
      reverse  : STRING          -> STRING
      add-back : SYMBOL # STRING -> STRING
      is-pal   : STRING          -> BOOLEAN
     end

   imports
     Symbols, Booleans, Naturals

   variables
     s1, s2, s3 : -> STRING
     sym1, sym2 : -> SYMBOL

   equations
  [E1]    equal(empty, empty) = true
  [E2]    equal(empty, sym2 ~ s2) = false
  [E3]    equal(sym1 ~ s1, empty) = false
  [E4]    equal(sym1 ~ s1, sym2 ~ s2) =
                      and( equal(sym1, sym2), equal(s1,s2))
  [L1]    length(empty) = zero
  [L2]    length(sym1 ~ s1) = S( length(s1) )
  [R1]    reverse(empty) = empty
  [R2]    reverse(sym1 ~ s1) = add-back(sym1,reverse(s1))
  [A1]    add-back(sym1, empty) = sym1 ~ empty
  [A2]    add-back(sym1, sym2 ~ s2) = sym2 ~ add-back(sym1,s2)
  [I1]    is-pal(s1) = equal(s1, reverse(s1))

end Strings


process module Palindrome-Behaviour
begin

  imports
    Booleans, Strings, Symbols

  atoms
    r : SYMBOL
    s : BOOLEAN

  processes
    PAL
    PAL : STRING

  variables
    w : -> STRING
   definitions
   PAL    = s(true).PAL + sum(x in SYMBOL, r(x).s(true).PAL(x ~ empty))
   PAL(w) = sum(x in SYMBOL, r(x).s(is-pal(x ~ w)).PAL(x ~ w))

end Palindrome-Behaviour


data module Ports
begin

  exports
    begin
      sorts
        PORT
      functions
        port-nr : NATURAL       -> PORT
        equal   : PORT # PORT -> BOOLEAN
        pred    : PORT          -> PORT
    end

  imports
    Naturals, Booleans

  variables
    x, y : -> NATURAL

  equations
  [E1]   equal(port-nr(x), port-nr(y)) = equal(x,y)
  [B1]   pred(port-nr(S(x))) = port-nr(x)
  [B2]   pred(port-nr(zero)) = port-nr(zero)

end Ports


process module Palindrome
begin

  imports
    Booleans, Symbols, Ports

  atoms
    r, s, c : PORT # SYMBOL
    r, s, c : PORT # BOOLEAN

  processes
    C : PORT
    C : PORT # SYMBOL
    C : PORT # SYMBOL # SYMBOL # BOOLEAN
    P

  sets
     of PORT
       CELL = PORT \ {port-nr(zero)}
     of atoms
       H = { s(p,s), r(p,s)  | p in CELL, s in SYMBOL } +
                 { s(p,b), r(p,b)  | p in CELL, b in BOOLEAN }
       I = { c(p,s)| p in CELL, s in SYMBOL } +
                 { c(p,b)| p in CELL, b in BOOLEAN }
    communications
      s(p,s) | r(p,s) = c(p,s) for p in CELL, s in SYMBOL
      s(p,b) | r(p,b) = c(p,b) for p in CELL, b in BOOLEAN

    variables
      x,y : -> SYMBOL
      v   : -> BOOLEAN
      i   : -> PORT

    definitions
      C(i)         = s(pred(i),true).C(i) +
                     sum(x in SYMBOL, r(pred(i),x).s(pred(i),true).C(i,x))
      C(i,x)       = sum(y in SYMBOL,
                        r(pred(i),y).sum(v in BOOLEAN, r(i,v).C(i,x,y,v))) +
                     sum(v in BOOLEAN,
                        r(i,v).sum(y in SYMBOL, r(pred(i),y).C(i,x,y,v)))
      C(i,x,y,v) = ( s(pred(i),and(equal(x,y), v)) || s(i,y) ). C(i,x)
      P            = hide(I,encaps(H,(merge( k in CELL, C(k)))))

  end Palindrome
```

figure 4.9 PSF specification of the palindrome recognizer.

In [109] it has been proven that P=PAL.

5. Considerations & Comparisons

5.1. A Comparison with Other Formal Description Techniques

In this section we will compare PSF with some other Formal Description Techniques. We will mainly focus on the comparison with LOTOS.

5.1.1. LOTOS

LOTOS (Language of Temporal Ordering Specification, [67]) is one of the two Formal Description Techniques, developed within ISO (International Organization for Standardization) for the formal specification of open distributed systems, in particular for those related to the Open Systems Interconnection (OSI) computer network architecture.

5.1.1.1. Similarities

Like PSF, LOTOS is a combination of two formalisms, namely a variation on ACT ONE [40] to describe data types and a process description part based on CCS [90]. As opposed to PSF, which was designed to be as close to ACP as possible, the distance between LOTOS and CCS is much greater. Many differences between LOTOS and PSF originate from the differences between ACT ONE and ASF, and CCS and ACP. We will start off with a list of constructions that are available in both languages:

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

figure 5.1 Similarities between LOTOS and PSF.

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

5.1.1.2. Action Prefix vs. Sequential Composition

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

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

figure 5.2 Action prefix vs. sequential composition.

5.1.1.3. Concurrency

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

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

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

LOTOS	*PSF*
B1 |[$g_1, \ldots, g_n$]| *B2*	encaps(*G*, *B1* || *B2*) where $G = \{g_1^*, \ldots, g_n^*, g_1^\wedge, \ldots, g_n^\wedge\}$ $g_i^* \mid g_i^\wedge = g_i$
B1 || *B2*	encaps(*A*, *B1* || *B2*) where $A = \{a_1^*, \ldots, a_n^*, a_1^\wedge, \ldots, a_n^\wedge\}$ $a_i^* \mid a_i^\wedge = a_i$ for all atomic actions in the alphabets of *B1* and *B2*.
B1 ||| *B2*	*B1* || *B2* where no atomic action from the alphabet of *B1* can communicate with any atomic action from the alphabet of *B2*.

figure 5.3 Concurrency constructs.

5.1.1.4. Communication

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

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

A gate can be coupled with one of these constructions so that the expression *g*?*x*:*integer* describes the set of all actions *g*<*v*> where *v* is an instance of sort *integer*.

LOTOS	*PSF*
g ! *a*($\underline{v}$)	*send*(*a*($\underline{v}$))
g ? *x*:*t*	sum(*x* in *t*, *receive*(*x*))

figure 5.4 Communication constructs.

5.1.1.5. Features Supported by LOTOS but not by PSF

There are some features in LOTOS that PSF (currently) does not support. Two of these features use *conditional constructs*. Conditional constructs are expressed as an equation between two value expressions or boolean expressions. In the former case, the condition is met if the two expressions evaluate to exactly the same value. Conditional constructs are used in synchronization as a *selection predicate* to impose a restriction on the values that may be transferred, and as *guards* in *guarded expressions*. PSF does not support conditional constructs, but has nevertheless the same expressive power. This is achieved by using *sets,* however we admit that this is carried out in a rather cumbersome way. In chapter 3 several extensions of PSF are considered.

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

figure 5.5 Conditional constructs and their translations in PSF.

Another feature that is not present in PSF is the *disabling* operator. The LOTOS expression: *B1* [> *B2*, means that as long as *B1* is active *B2* can take over the execution at any time, resulting in the disappearance of *B1*.

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

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

In PSF there is no such thing as value passing. All processes coexist and exchange information by communication, although a lot of them may be held up, waiting to take part in a communication. The *chaining* operator, which is merely an abbreviation of two renamings, a merge and an abstraction, in ACP [101] resembles the enabling operator but has a slightly different semantics.

5.1.1.6. Data Specification

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

5.1.1.7. Modularization

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

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

figure 5.6 Example of a LOTOS specification.

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

5.1.2. Estelle

Estelle [66] is the other Formal Description Technique developed by the ISO. Estelle is based upon an extended finite state machine model.

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

A specification in Estelle consists of a set of modules which can communicate with each other. Modules represent finite state machines and are defined by using a number of primitives which are extensions to ISO Pascal. So a specification in Estelle looks like a Pascal program with some extra facilities.

Being based on Pascal, there are no abstract data types and verification of specifications is hindered.

5.1.3. COLD

COLD [41] is a series of languages developed in the framework of ESPRIT project 432 (METEOR). COLD is defined by means of a translation of its grammatical constructs to the constructs of a three layered formal language. The top layer of this kernel is a special version of lambda calculus, which is called $\lambda\pi$, and is used for modelling parameterization. Expressions in this lambda calculus contain terms from a special many-sorted algebra, called CA, which is used for modelling modularization constructs. This algebra constitutes the middle layer. The constants used in the terms of this algebra are presentations of logical theories. The logical language used at the bottom level is based on a special infinitary logic, called MPL_{ω}. Every construct in a COLD specification corresponds to an expression in the kernel of formal languages with a well-defined semantics. COLD specifications are translated by means of attribute grammars to the kernel.

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

5.2. A Comparison with Parallel Programming Languages

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

5.2.1. Extended Programming Languages

By extended programming languages we mean languages that have been extended afterwards to include features to support concurrent programming. Two examples of these languages are *Concurrent Pascal* [32] and *Concurrent Euclid* [63]. Both languages have some extra features, such as *processes* to define concurrently executable pieces of the program and *monitors* to guarantee (mutual) exclusive access to variables. Communication between processes is established by means of shared variables.

5.2.2. Modula, CHILL, Ada

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

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

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

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

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

A feature that is lacking in PSF but present in Ada is: *time*. It is possible in Ada to delay a process for a while by means of the *delay* statement. In the context of a *select* statement, that implements the idea of choosing non-deterministically between guarded commands (see [38]), a *delay* develops into a *time-out*, when used as a *guard*. This means that only after a certain period of time one branch of a select statement becomes active, i.e. the guard becomes true, when all the guards of the other branches remain false. There is strong need for such mechanisms in real-time applications. The notion of real time was introduced in ACP in [6].

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

- Variables.
  A major component of a computer is the memory, which comprises a large number of *memory cells*. The reflection of these memory cells in a programming language are variables.
- Assignment Operation
  Closely tied to the memory architecture is the notion that everything that has been computed must be stored, i.e., assigned to a cell. This accounts for the assignment operation in imperative programming languages.
- Repetition
  A program in an imperative language usually accomplishes its task by executing a sequence of elementary steps repeatedly. The von Neumann architecture forces this way of solving problems.

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

All three languages mentioned, support some kind of *information hiding* by allowing the definition of *abstract data types*. This is achieved by defining a data type and some functions that operate on this type that are grouped together in some construct. The representation of the abstract data types and the implementation of the functions is defined within this construct, but is hidden for the outside world. In this way the outside world gets only an abstract view of the data types involved. Information hiding is provided by *modules* in Modula and CHILL and by *packages* in Ada. In PSF, data abstraction and procedural abstraction is provided by *data* and *process modules* and the *import* and *export* constructions. There are some differences however. In PSF an imported object automatically appears in the export section of the importing module. This is not the case in any of the three programming languages. Nor is there something like the *origin rule* as in PSF, allowing multiple imports of the same module.

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

5.2.3. POOL 2

POOL (Parallel Object-Oriented Language) [2] is a programming language designed to integrate object oriented programming with parallelism. All objects in a POOL program may execute in parallel, so this resembles the notion of processes in PSF.

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

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

In [107] we can find a study of how to implement ACP specifications in POOL and a more extensive comparison between POOL and ACP.

5.2.4. Occam 2

Occam [65] is a programming language based on CSP [61]. It has been designed by INMOS and serves as a programming language for the INMOS transputer, which in turn can be considered an Occam machine. The transputer is a single processor with some internal memory and has four channels with which it can be connected to neighbouring transputers. It is expected that a set of such transputers will form an easily extendible parallel computer.

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

- assignment
- input
- output

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

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

5.2.5. PARLOG

PARLOG [34] is a parallel logic programming language that is characterized by the use of the concepts of guards and committed choice nondeterminism as in Dijkstra's procedural language of guarded commands [38]. It is a member of a family of languages that further consists of Concurrent Prolog [96] and GHC [99].

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

PARLOG does not provide any means for specifying abstract data types, all data types have to be represented by the programmer, by means of *lists*, nor does it incorporate any modularization concepts. PARLOG is of interest however, because it has been used to translate LOTOS specifications into an executable PARLOG program [45].

5.3. Current State and Future Developments

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

5.3.1. Tools

The structure of the PSF toolkit under development is displayed in figure 5.7. On top is the PSF formalism and at the centre is the Tool Interface Language as described in chapter 4. The lower part contains the tools.
The compiler from PSF into TIL consists of a parser, a library manager and a normalizer. Starting from a modular PSF specification and a library of previously compiled modules they produce a flat specification in TIL. This specification is used as input for the tools.

The simulator enables the user to walk through a process by displaying its behaviour step by step. The user can control the execution by choosing one of the alternatives in case of a non-deterministic choice. In order to provide long test-runs, the choice of the alternative can be delegated to a random generator. Breakpoints may be used to control this process.

The proof assistant makes it possible to interactively manipulate process expressions using a collection of axioms and some built in tactics. A proof of equivalence of two processes, by means of a transformation of one into the other, can be seen as a verification that the given process obeys a specified behaviour.

Algebraic specifications are implemented by interpreting them as term rewriting systems. The term rewriting tool is a general tool, which may be used by the other tools. It can also be used on its own for testing algebraic specifications.

Other tools, for example for deciding bisimulation equivalence, and interfaces to external tools are under development.

The tools are being developed and are executed on a SUN workstation under the UNIX operating system using the windows system X. All programs are written in the programming language C and tools that come with UNIX, such as Lex and Yacc. This guarantees both performance and portability.

An overview of the system is in [104] and parts of the toolkit are described in more detail in [103] and [88].

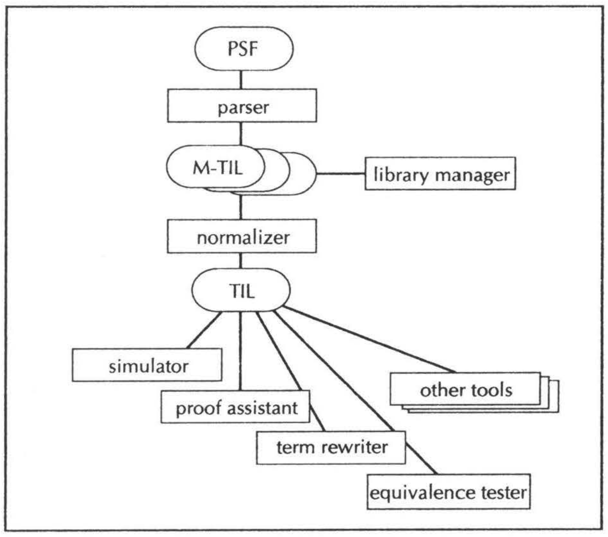


figure 5.7 The PSF Toolkit

5.3.2. Comparison with Other Tools

Although the subject of this thesis is not the PSF toolkit, we will indicate briefly how it relates to tools from the same area of application.

The first step in concurrent system design is the construction of a specification. The input formats for the various tools range from ad hoc input formats to standardized languages such as LOTOS [67] and Estelle [66]. Some tools support graphical or hierarchical design [43]. The main input language of the PSF toolkit is, of course, PSF. However, the construction of the toolkit allows for any input language to be used, as long as it is based on ACP-like process specifications and algebraic data type specification. Two experiments with other input formats show this. The first experiment is the tabular viewer for specifications [93], which indicates how a more module based design can be supported, and the second experiment is the construction of a compiler from the language XP [105] to TIL. An interesting development is the design of common formats, which allow independent tools to communicate with each other [78].

Simulation of the behaviour of a specification is the basic way to test for possible errors. Most tools for FDT's are provided with such a simulation mode. Some tools, like the PSF toolkit, allow only for textual simulation, but others also support graphical animation of the specified system [79].

Most tools for concurrent system verification are based on finite systems. This means that the involved processes are represented by finite structures, such as transition graphs. Two transition graphs can be tested for equality with respect to some equivalence relation. Tools based on this approach are described in [35], [79] and [42]. In general, these methods suffer from the, so called, state explosion problem.

The PSF toolkit, however, focuses on ACP-style verification. This means that concurrent systems are verified by algebraic manipulation of process expressions. Another system which uses this approach is the process algebra manipulator PAM [77]. This tool allows users to define their own calculi. Other verification tools also include techniques such as model checking [75].

In our opinion, the most useful approach towards computer aided verification will show to be a combination of the approaches listed above. Algebraic manipulation is used to split a proof into a number of sub-proofs, each of which can be delegated to an automatic verification tool based on equivalence testing.

5.3.3. The Language

Though PSF as presented in this report is already a rather powerful specification language, yet we are thinking of some enhancements. There are still some ACP constructions that have not been implemented in PSF. These constructions are, e.g. chaining, (dynamic) process creation, renaming (of atoms), interrupts, priorities and mode transfer. We will have to examine which constructions can be incorporated in PSF without affecting the semantical model as yet defined. Chapter 3 describes a number of possible extensions of PSF.

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

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

The choice of algebraic specification techniques for the description of the data types may be subject of discussion. Clearly, without additional features such as modularization and special constructs, algebraic specification is too weak for the specification of large software systems. See e.g. [28] for a discussion of the complexity of defining finite sorts with equality. However, we believe that in the context of process specification, the data types which are needed are of a quite simple nature. Several case studies have indicated that algebraic specification is appropriate for the data types which were involved. Furthermore, the specification language will benefit from the clear and simple semantics of algebraic specifications and the elegant way of prototyping them by means of a term rewriting system.

The semantics of the modularization concepts from PSF are defined by means of a normalization procedure which is quite complex. A more algebraic approach, such as in the module algebra [21], would be cleaner.

Another point of discussion if the fact that imported objects automatically belong to the external signature of the importing module. In some cases this contradicts the principle of information hiding, so an explicit hiding mechanism of objects will show to be useful.

5.4. Conclusions

In this chapter we have presented PSF, a new formalism to describe process behaviour. We have shown that it is possible to integrate a formal approach towards data types in this formalism, as opposed to the informal way in which data types are generally treated in ACP. We hope that PSF will be a contribution to the construction of more reliable software for concurrent systems.

Chapter 3

EXTENDING PSF

Extensions of PSF (a Process Specification Formalism) are proposed. These extensions include facilities for making conditional choices, operators for disabling, interrupts and priorities, and constructions for state manipulation.

1. INTRODUCTION

The specification language PSF is developed to facilitate the specification and verification of parallel systems using the concurrency theory ACP. Whereas PSF only supports the basic features of ACP, many extensions of ACP have been proposed, such as priorities [7], the state operator [5], renamings [5] and process creation [18]. In this chapter suggestions are made on how to add new features to PSF, in order to be able to make more concise and realistic specifications.

The problem with extending an existing language is that every extension leads to a new language. Tools have to be rewritten and special care has to be taken to keep the new version downwardly compatible, in the sense that old specifications remain correct. Extensions of PSF can take place at three levels. At the level of the modular structure, at the level of the specification of data types and at the level of process description. Since this chapter studies the embedding of extensions of ACP into PSF, we will focus on the last level. We try to develop the extensions in such a way that the structure of a module remains unchanged, and that no special data types or functions are needed. This way, modifications are localized to the process specification part of PSF.

Of course we have to cope with the problem that most extensions of ACP deal with additional data structures. The priority operator for example needs a partial ordering on the atomic actions and the state operator is defined using the notion of a state and functions acting on states (the so called *action* and *effect* functions). As a result of the requirement that only new process operators may be introduced, the need emerges for a restyling or simplification of these extensions of ACP.

In this chapter we give three examples of how to extend PSF. The first one introduces conditional choices, a mechanism to specify the flow of control more easily. A simple predecessor of this operator is the guarded command operator from [11], a feature which is also present in the LOTOS specification language [67]. The second example shows how to handle interrupts and disabling of processes. Disruption, which is also present in LOTOS, was introduced into ACP in [17]. In order to handle interrupts directly after being raised, a priority operator is introduced, which is based on the θ operator from [7].

Finally, the extension of PSF with an explicit notion of a state is considered. For this purpose, so called state variables are introduced, which behave similarly to variables in an imperative programming language. A related mechanism, the state operator, was introduced into ACP in [5]. An application of the use of state variables is given, where they are used to model asynchronous communication between processes.

2. Conditional Choices

2.1. General

In PSF there exists no explicit mechanism that, depending on the value of some data object, determines the control flow of a process. This can only be done by introduction of a new process, which has that data element as index. Choices are made by adding for each condition a new entry in the list of process definitions, followed by the appropriate actions. This way auxiliary process names are needed only to control choices.

As an example look at the following specification of a buffer. This PSF fragment defines the process *Buffer*, indexed with its contents, a queue of elements from some data type *D*. The behaviour of this buffer depends on its contents. We presume a data type *Queues* given, having appropriate definitions of the functions which are used in this example.

```
Buffer(empty-queue) =
      sum(d in D, input(d)  . Buffer(add-back(d, empty-queue)))
Buffer(add(e,q)) =
      sum(d in D, input(d)  . Buffer(add-back(d, add(e,q)))) +
      output(e)  . Buffer(q)
```

A more concise specification could be obtained using a *case* construction.

```
Buffer(q) = sum(d in D, input(d) . Buffer(add-back(d, q))) +
             case is-empty(q) = false
               do output(top(q)) . Buffer(pop(q)) od
```

The option to send an item from the buffer to the output is only enabled if the queue is not empty.

The condition in a *case* construction with one alternative is an equation of terms of the same sort. If the two terms are equal, the expression between *do* and *od* will be enabled. If the two terms are not equal the expression equals *deadlock*.

The following example shows the use of a conditional choice with more alternatives. It defines the process *Distribution,* which plays a role in interpreting signals from a remote control of a television set. The incoming messages are distributed over the various components, which will handle the messages.

```
Distribution =
   sum(m in messages, receive(m) .
       case m
       = volume-up, volume-down           do s(sound-control, m)    od
       = brightness-up, brightness-down do s(display-control, m) od
       = optimal                          do s(sound-control, m) .
                                             s(display-control, m) od)
```

The semantics of the case operator are given by the following laws. We use the auxiliary guarded command operator as defined in [11]. The definition of the guarded command is given in the second part of the following table. Let s and $t_{k,l}$ ($1 \le k \le n$, $1 \le l \le i_k$, $i_k \ge 1$, $n \ge 1$) be data terms over some given signature. Let x_i ($1 \le i \le n$) be processes. We require that the left and right hand-side of the conditions are of the same type.

Axioms GC1 and GC2 can be used to eliminate the guarded command operator if the guard can be evaluated. Axioms GC3 and GC4 allow logic manipulation of formulas. Substitution of variables is handled by GC5. GC6 up to GC13 deal with the combination of the guarded command with other operators.

From the definitions it follows that the case operator can be eliminated from closed process expressions, under the assumption that the conditions can be decided and the expression does not contain the generalized sum or merge construct.

CASE	**case** s	=	
	$= t_{1,1},\ldots,t_{1,i_1}$ **do** x_1 **od**		$(s=t_{1,1}) :\rightarrow x_1 + \ldots + (s=t_{1,i_1}) :\rightarrow x_1 +$
	$= t_{2,1},\ldots,t_{2,i_2}$ **do** x_2 **od**		
	...		$\ldots +$
	$= t_{n,1},\ldots,t_{n,i_n}$ **do** x_n **od**		$(s=t_{n,1}) :\rightarrow x_n + \ldots + (s=t_{n,i_n}) :\rightarrow x_n$

GC1	$\phi \Rightarrow (\phi :\rightarrow p = p)$
GC2	$\neg\phi \Rightarrow (\phi :\rightarrow p = \delta)$
GC3	$\phi :\rightarrow (\psi :\rightarrow p) = (\phi \wedge \psi) :\rightarrow p$
GC4	$(\phi \vee \psi) :\rightarrow p = (\phi :\rightarrow p) + (\psi :\rightarrow p)$
GC5	$(v{=}t) :\rightarrow p = (v{=}t) :\rightarrow p[v{:=}t]$
GC6	$\phi :\rightarrow (x \cdot y) = (\phi :\rightarrow x) \cdot y$
GC7	$\phi :\rightarrow (x + y) = (\phi :\rightarrow x) + (\phi :\rightarrow y)$
GC8	$(\phi :\rightarrow x) \,⫽\, y = \phi :\rightarrow (x \,⫽\, y)$
GC9	$(\phi :\rightarrow x) \mid y = \phi :\rightarrow (x \mid y)$
GC10	$x \mid (\phi :\rightarrow y) = \phi :\rightarrow (x \mid y)$
GC11	$\partial_H(\phi :\rightarrow x) = \phi :\rightarrow \partial_H(x)$
GC12	$\tau_I(\phi :\rightarrow x) = \phi :\rightarrow \tau_I(x)$
GC13	$\phi :\rightarrow (x \cdot y) = (\phi :\rightarrow x) \cdot (\phi :\rightarrow y)$

table 1 Algebraic laws for Conditional Choice

2.2. Transition Rules

Since the conditional choice operators can be viewed as a shorthand notation, we only give the operational semantics for the guarded command.

GC1	$\dfrac{x \xrightarrow{a} x',\ \phi}{(\phi :\rightarrow x) \xrightarrow{a} x'}$	GC2	$\dfrac{x \xrightarrow{a} \surd,\ \phi}{(\phi :\rightarrow x) \xrightarrow{a} \surd}$

table 2 Transition rules for Conditional Choice

3. Interrupts and Disabling

In this section three new operators are introduced in PSF. The priority operator is used to give some actions higher priority than others, e.g. actions denoting an interrupt. The interrupt and disruption operator are used to express the possibility that a process is always willing to perform a special interrupt or disabling action.

3.1. PRIORITIES

3.1.1. General

In [7] an interrupt mechanism for ACP was introduced. This mechanism used the priority operator θ for filtering out all actions with highest priority. The hierarchy of atomic actions is defined by a fixed partial ordering. We introduce a new operator which has as a parameter a set of atomic actions that have priority over all other actions. Thus if an expression has an alternative that starts with an atom with high priority, alternatives that start with an atom with low priority will be suppressed. If the expression has no alternatives starting with an atom with high priority, all alternatives are still enabled.

So for $c \neq a$ and $c \neq b$, the expression *prio({a}, a + b)* equals *a* and *prio({c}, a + b)* equals *a + b*. Note that the priotity operator is not monotonic with respect to the alternative composition.

In this way we can define the two levels of high and low priority, but by repeatedly applying the *prio* operator we can introduce more levels. The innermost set defines the atoms with highest priority, as is shown by the following examples.

$prio(\{a,b\}, prio(\{c\}, a + b + c + d)) = prio(\{a,b\}, c) = c$, while

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

3.1.2. Semantics

The semantics of the *prio* operator are given by the following equations. The atomic action *a* may not be equal to *skip* and the set *S* may not contain *skip*.

PRI1	$prio(S, x) = x \lhd_S \delta$	
PRI2	$a \lhd_S b = a$	if $a \in S \vee b \notin S$
PRI3	$a \lhd_S b = \delta$	otherwise
PRI4	$a \lhd_S \delta = a$	
PRI5	$\delta \lhd_S x = \delta$	
PRI6	$a \lhd_S skip = a$	
PRI7	$skip \lhd_S x = skip$	
PRI8	$x \lhd_S y.z = x \lhd_S y$	
PRI9	$x \lhd_S (y+z) = (x \lhd_S y) \lhd_S z$	
PRI10	$x.y \lhd_S z = (x \lhd_S z).(y \lhd_S \delta)$	
PRI11	$(x+y) \lhd_S z = (x \lhd_S y) \lhd_S z + (y \lhd_S x) \lhd_S z$	
PRI12	$x \lhd_S (\phi :\text{->} y) = \phi :\text{->} (x \lhd_S y) + \neg\phi :\text{->} (x \lhd_S \delta)$	
PRI13	$(\phi :\text{->} x) \lhd_S y = \phi :\text{->} (x \lhd_S y)$	

table 3 Algebraic laws for the priority operator

In the definition of the priority operator we use an auxiliary operator $\triangleleft_S$. This operator is parameterized with a set of actions. It serves to calculate the context of the first actions of its left-hand side. This context, that is the collection of all alternative actions, is being built up in the right-hand side. If some action has low priority and its context can do an action with high priority, this low priority action is blocked. From rule PRI12 it follows that we need negation of guards if we combine priorities with the guarded command.

The θ operator in a process expression can be replaced by a finite series of *prio* operators if two conditions are met. The first one is that the ordering of the atoms must be total and the second condition is that the number of classes in the equivalence relation induced by the total ordering must be finite. Of course in the setting of PSF, every class, except for the one with lowest priority should be expressible using the operators defined in PSF to construct sets.

Conversely, it is always possible to replace a *prio* operator, or a series of them, by a θ with appropriate ordering on the atomic actions. This ordering however should be a parameter of the θ operator and not of the specification as a whole, as proposed in [7].

3.1.3. Transition Rules

The transition rules for the *prio* operator are straightforward. A process can perform a certain action unless it can do an action with higher priority. The notation $x \overset{b}{\not\rightarrow}$ is used to indicate a negative condition. It means that process x cannot do a b-transition to $\surd$ or another process.

prio1	$\dfrac{x \xrightarrow{a} x',\ a \in S}{\mathrm{prio}(S,x) \xrightarrow{a} \mathrm{prio}(S,x')}$	prio2	$\dfrac{x \xrightarrow{a} \surd,\ a \in S}{\mathrm{prio}(S,x) \xrightarrow{a} x'}$
prio3	$\dfrac{x \xrightarrow{a} x',\ a \notin S,\ \forall_{b \in S}\ x \overset{b}{\not\rightarrow}}{\mathrm{prio}(S,x) \xrightarrow{a} \mathrm{prio}(S,x')}$	prio4	$\dfrac{x \xrightarrow{a} \surd,\ a \notin S,\ \forall_{b \in S}\ x \overset{b}{\not\rightarrow}}{\mathrm{prio}(S,x) \xrightarrow{a} \surd}$

table 4 Transition rules for the priority operator

Using techniques from [50] it can be shown that these rules, which involve negative premises, in combination with the rules of chapter 2, section 3.7, define a transition relation.

3.2. PRIORITIES, INTERRUPTS AND DISABLING

3.2.1. General

The priority operator introduced above enables us to force the action with highest priority to be performed. For modelling interrupts and disabling however, this operator is not enough. A process that is able to accept an interrupt should be willing to accept this interrupt at any instant. After every execution of an atomic action the alternative to do the interrupt action must be present.

If we want to extend the process expression *a.b.c.d.e* so that it can be disabled by the action *i* we would have to add this option at any position:

$i + a(i + b(i + c(i + d(i + e))))$.

Adding an interrupt *i* followed by interrupt handler *I* costs even more overhead. It would result in the following system of equations:

$x_1 = i.I.x_1 + a.x_2$
$x_2 = i.I.x_2 + b.x_3$
$x_3 = i.I.x_3 + c.x_4$
$x_4 = i.I.x_4 + d.x_5$
$x_5 = i.I.x_5 + e$

It is useful to have a shorthand for these constructions. We use the mode transfer operator from [17] to handle disruption and we define a new operator to handle interrupts. We will not use the same notational convention as in [17]. The expression *dis(x,y)* is used to express the fact that process *x* can be disrupted at any time by process *y*. If *y* is called and is finished, then the whole process is finished. The expression *int(x,y)* means that process *x* can be interrupted at any time by process *y*. After *y* has finished, *x* resumes. Both *int(x,y)* and *dis(x,y)* finish if *x* finishes.

3.2.2. Semantics

The semantics for disruption and interrupts are given by the following algebraic laws. In the definition of the interrupt operator we need the auxiliary *delayed interrupt* operator, which is denoted by *dint*. This operator behaves exactly as the interrupt operator, with the restriction that it cannot start with the interrupting process. The second extra operator is called *enable*. The first argument can only be executed if the second one is not equal to deadlock.

INT	int(x, y) = dint(x, y) + enable(y.int(x, y)),x)
DINT1	dint(a, x) = a
DINT2	dint(a.x, y) = a.int(x, y)
DINT3	dint(x+y, z) = dint(x, z) + dint(y, z)
DINT4	dint(δ, x) = δ
DINT5	dint(ϕ:–>x, y) = ϕ:–>dint(x, y)
EN1	enable(x, a) = x
EN2	enable(x, y.z) = enable(x, y)
EN3	enable(x, y+z) = enable(x, y) + enable(x, z)
EN4	enable(x, δ) = δ
EN5	enable(x, ϕ:->y) = ϕ:->enable(x, y)

DIS1	dis(a, x) = a+x
DIS2	dis(a.x, y) = a.dis(x, y) + y
DIS3	dis(x+y, z) = dis(x, z) + dis(y, z)
DIS4	dis(δ, x) = δ
DIS5	dis(ϕ:–>x, y) = ϕ:–>dis(x, y)

table 5 Algebraic laws for disruption and interrupts

Axioms DIS1 to DIS4 are from [17]. The equation INT is of another nature than the rest of the laws. It should be interpreted as follows. The process *int(x,y)* is a solution of the recursive specification

P = dint(x,y) + enable(y.P,x)

If specifications are guarded, existence and uniqueness of a solution is provided by the Recursive Definition Principle and the Recursive Specification Principle (see [24] or [14] for definitions of these principles). It is easy to see that if the *int* operator is used in a guarded recursive specification, the definition of this operator with rule INT will also be a guarded specification.

We have made the choice that *int(x,y)* is an infinite process, even if x and y are finite. This is motivated by the idea that an interrupt can occur an unspecified number of times before the interrupted process is granted time to resume its normal operation.

The reason for introducing the delayed interrupt operator is that simply setting

int(x+y, z) = int(x,z) + int(y,z)

would result in

int(a+b, c) = int(a,c) + int(b,c) = a + c.int(a,x) + b + c.int(b,x)

instead of

int(a+b, c) = dint(a+b, c) + c.int(a+b, c) = a + b + c.int(a+b,x)

The first expression implies that the choice between a and b can be forced by executing one of the two possible c actions.

An interpretation of the disruption operator in the graph model is given in [17]. Let G be the graph of process x and let H be the graph of process y. We may assume that the root of H has no incoming edges (see e.g. [14] for a rootunwinding procedure which preserves bisimulation equivalence). The graph of *dis(x,y)* is now constructed by taking the disjoint union of G and H. For every transition with label a from the root of H to node h of H add to every non-terminal node of G a transition with label a to h.

The interrupt operator can be interpreted similarly. The graph of *int(x,y)* is constructed from the graphs of x and y by creating for every non-terminal node g of G a disjoint copy H_g of H. Then identify the root of H_g and the terminal nodes of H_g with node g.

We can easily show that the domain of finite and acyclic process graphs is not closed under application of the interrupt operator. The process *int(a,b)* has an infinite trace b^{ω} and thus is not finite. The class of regular process graphs however is closed under application of the interrupt operator. This holds because if process *x* has *n* states and process *y* has *m* states, the construction of *int(x,y)* yields at most *n*m* states.

As a consequence we do not have an elimination theorem for finite process expressions, which states that every finite expression without variables can be rewritten in an equivalent expression which is only built up of atomic actions, sums and prefix multiplication. We do have an elimination theorem for processes defined by means of systems of guarded recursive equations. Every process defined by a guarded recursive specification involving the *int* operator can be defined by a specification without the *int* operator. This specification can be obtained from the original one by using the algebraic laws for the *int* operator. The proof is by induction on the structure of the expressions.

Note that in contrast with the disruption operator, the interrupt operator is not associative for closed terms. The process *int(int(a,b),c)* has a trace *ca*, while *int(a,int(b,c))* does not.

3.2.3. Transition Rules

dis1	$\dfrac{x \xrightarrow{a} x'}{dis(x,y) \xrightarrow{a} dis(x',y)}$	int1	$\dfrac{x \xrightarrow{a} x'}{int(x,y) \xrightarrow{a} int(x',y)}$
dis2	$\dfrac{x \xrightarrow{a} \surd}{dis(x,y) \xrightarrow{a} \surd}$	int2	$\dfrac{x \xrightarrow{a} \surd}{int(x,y) \xrightarrow{a} \surd}$
dis3	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} y'}{dis(x,y) \xrightarrow{b} y'}$	int3	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} y'}{int(x,y) \xrightarrow{b} y'.int(x,y)}$
dis4	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{dis(x,y) \xrightarrow{b} y'}$	int4	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} y'}{int(x,y) \xrightarrow{b} y'.int(x,y)}$
dis5	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} \surd}{dis(x,y) \xrightarrow{b} \surd}$	int5	$\dfrac{x \xrightarrow{a} \surd \quad y \xrightarrow{b} \surd}{int(x,y) \xrightarrow{b} int(x,y)}$
dis6	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \surd}{dis(x,y) \xrightarrow{b} \surd}$	int6	$\dfrac{x \xrightarrow{a} x' \quad y \xrightarrow{b} \surd}{int(x,y) \xrightarrow{b} int(x,y)}$

table 6 Transition rules for interrupt and disabling

3.3. An Example

As an example of the use of these operators we specify a skeleton of a very simple operating system. This operating system is inspired by the Commodore 64 basic operating system [36]. Only the top-level operations are specified. Other processes and data types are assumed to be imported.

If the computer is switched on it does a *ColdStart*. After doing a memory test the *HotStart* sequence is activated, which resets all jump vectors (*MemTest* and *ResetVectors* are imported processes). Then the Normal-Operation starts which has the possibility to either be disrupted by a *reset*-signal, which forces a *HotStart*, or it can be interrupted by an *alarm* from the clock. This *alarm* indicates that it is time to execute the regular interrupt routine, which scans both keyboard and serial port for activities. If one of them has incoming data, this will be read and stored in the keyboard buffer and serial port buffer. After an interrupt, normal operation is resumed, which consists of reading and interpreting tokens from the user program stored in memory (*GetNextToken* and *Interpret* are imported processes).

The overall system consists of the processor, initialized with a *ColdStart*, together with a number of devices which are not specified in this example. An *alarm* action has priority over all other actions, except for a *reset*, which has highest priority.

```
process module C64
begin

  exports
    begin
      processes
        ColdStart, HotStart, Normal-Operation, Interrupt-Sequence,
        ScanKeyBoard, ScanSerialPort, Reset-Sequence, System
      atoms
        ...
    end

   sets of atoms
    IntSet = {alarm}
    DisSet = {reset}

  communications
    rec-alarm | send-alarm = alarm
    rec-reset-signal | send-reset-signal = reset

  imports
    ...

definitions
ColdStart = MemTest . HotStart
HotStart = ResetVectors .
             dis(
               int(Normal-Operation, Interrupt-Sequence),
             Reset-Sequence)
Normal-Operation = sum(t in token, GetNextToken(t) . Interpret(t)) .
                   Normal-Operation
Interrupt-Sequence = rec-alarm . ScanKeyBoard . ScanSerialPort
```

```
ScanKeyBoard =
    sum(b in BOOL, key-pressed(b) .
    case b
      = true    do sum(k in key, get-key(k)  . put(KbdBuffer, k)) od
      = false   do skip od)
ScanSerialPort =
    sum(b in BOOL, data-arrival(b)  .
    case b
      = true    do sum(d in data, get-datum(d)  .
                                        put(SerBuffer, d)) od
      = false   do skip od)
Reset-Sequence = rec-reset-signal  . HotStart

System = prio(DisSet, prio(IntSet,
             ColdStart || KeyBoard || SerialPort || Display || Clock))
end C64
```

4. STATES

4.1. GENERAL

The explicit notion of a state and functions controlling the state of a process were introduced in ACP in [5]. With the state operator the functional approach to specifying in ACP was enriched (or polluted as some say) with imperative aspects.

In spite of these discussions, in many cases explicit manipulation of states can be very useful. In [108], for example, it was demonstrated that an expression like the following seems to be very natural.

$$(\sum_x r1(x) \mid\mid \sum_y r2(y) \mid\mid \sum_z r3(z)) \,.\, s4(x+y+z)$$

The intention is to read in three values, in an order which is immaterial, followed by an operation depending on these values. Plausible as this seems, the scope rules for the variables bound by the *sum* constructs are disobeyed. Thus the x, y and z variables from the atomic action $s4(x+y+z)$ are not bound, making this an illegal expression.

In [3] this problem was resolved by expanding the merge to an expression containing every order of execution of the read actions, with the use of the ACP axioms. Due to the exponential growth of the length of such an expression, this is not satisfactory. The state operator (as defined in [5]) offers a simple solution to this problem by explicitly adding the values read in to the state of the process, and making it possible to inspect and use the values at any instant.

Adding the state operator to PSF would imply that also *action* and *effect* functions should be defined. These are functions acting on states and atoms, thus adding this operator imposes a number of predefined data types and functions on them. Since this is not in accordance with the view on extending PSF exposed in the introduction, we will propose a simpler construction with implicitly defined action and effect functions.

In [106] also a variation on the state operator is described. This operator is called the *register operator*. It considers an infinite number of registers, labeled with the natural numbers, each of which can contain a value of some data domain. Every domain is enriched with a special value *undefined*, indicating that a register contains no value.

The approach taken here makes use of *state variables*, which are similar to these registers, but are more tailored to PSF.

4.2. State Variables

In addition to the static type of variables that already exists in PSF, we introduce state variables, whose value can change dynamically. The static variables in a process expression are fixed at binding time, and serve as a sort of shorthand. So if a static variable once has been assigned a value, it will remain unchanged within the entire scope of the variable.

The value of a state variable may change during "execution" of the process by using an assignment action.

4.2.1. Basics

Let V be a collection of state variables. To every variable we assign a type. We extend the collection of data terms in a straightforward way by allowing data terms to contain state variables in a correctly typed manner. This collection of extended data terms is called D_V. Likewise, the collection of atomic actions is extended to A_V by allowing extended data terms as index. Furthermore assignments of the form *[v:=t]* will be considered atomic actions, where *v* is a state variable and *t* is an extended data term of the same type. The extended collection of atomic actions is called $A_{V,Ass}$. Except for the left hand-side of an assignment and in a declaration, state variables are always referred to by placing the name within square brackets.

The *var* operator serves to introduce state variables in a process expression. This operator declares the name and type of a state variable, and optionally gives the variable an initial value. In the following example we define a process which declares a variable *v* of sort *D*, assigns the value d_0 to it, performs action $a(d_0)$, assigns value $f(d_0)$ to *v* and executes action $a(f(d_0)*f(d_0))$.

var(v in D, [v:=d_0] . a([v]) . [v:=f([v])] . a([v]*[v]))

Using initialization of v, the example looks as follows:

var(v:=d_0 in D, a([v]) . [v:=f([v])] . a([v]*[v]))

Note that these examples are similar, but they do not represent the same process. This is because the assignment in the first example is an atomic action, while the initialization in the second example is not. The first expression will equal the following:

skip . a(d_0) . skip . a(f(d_0)*f(d_0)),

while the second expression equals

a(d_0) . skip . a(f(d_0)*f(d_0)).

After "execution" an assignment becomes the internal action skip.

4.2.2. Semantics

We use the special symbol ↑ to indicate that a state variable has no value assigned yet. For conciseness, in the following tables we will write *var(v:=↑ in D, x)* instead of *var(v in D, x)*.

Let *v* and *w* (*v≠w*) be state variables of type *D* and *D'* respectively, and let *r*, *s* and *t* be data terms of type *D'*, *D* and *D* respectively, or let *r*, *s* and *t* denote the special symbol ↑. For data terms *s* and *t* the expression *t([v]/s)* is the data term *t* in which all occurrences of *[v]* are replaced by *s*. We define *t([v]/↑)* by ↑ if *[v]* occurs in *t* or by *t* if *[v]* does not occur in *t*. Likewise for an atomic action *a* from A_V we define *a([v]/s)* as the action in which all occurrences of *[v]* are replaced by *s* and *a([v]/↑)* by δ if *[v]* occurs in *a* or by *a* if this is not the case. Furthermore α is an element of $A_{V,Ass}$.

VAR1	var(v:=s in D, x) = var(v':=s in D, x([v]/[v']))	if [v'] not in s and not in x
VAR2	var(v:=s in D, a) = a([v]/s)	
VAR3	var(v:=s in D, [v:=t]) = skip	
VAR4	var(v:=s in D, [w:=r]) = [w:=r([v]/s)]	
VAR5	var(v:=s in D, a.x) = a([v]/s) . var(v:=s in D, x)	
VAR6	var(v:=s in D, [v:=t].x) = skip . var(v:=t([v]/s) in D, x)	
VAR7	var(v:=s in D, [w:=r].x) =	
	[w:=r([v]/s)]. var(v:=s in D, x)	if [w] not in s
	sum(d in D', [w]=d :-> [w:=r([v]/s)] . var(v:=s([w]/d) in D, x)	otherwise
VAR8	var(v:=s in D, x+y) = var(v:=s in D, x) + var(v:=s in D, y)	
VAR9	var(v:=s in D, δ) = δ	
VAR10	var(v:=s in D, φ:–>x) = φ([v]/s):-> var(v:=s in D, x)	
VAR11	a([v]) = sum(d in D, [v]=d :-> a([v]/d))	
VAR12	∇(H, [v:=s]) = [v:=s]	∇ is encaps or hide
VAR13	∇(H, [v:=s].x) = [v:=s] . ∇(H, x)	∇ is encaps or hide
VAR14	[v:=s] | α = δ	
VAR15	[v:=t] $\lhd_S$ x = [v:=t]	
VAR16	a $\lhd_S$ [v:=t] = a	

table 7 Algebraic laws for state variables

Notes

- Since the collection of atomic actions is extended, we have to reformulate all algebraic laws for other process operators, which involve atomic actions. In most of the cases the axioms hold for extended atoms and assignments as well. This is not the case if a closed action is needed because membership of a set has to be tested, such as in the axioms for the encaps and the hide operator. Axiom VAR11 is used in these cases to obtain closed atomic actions in expressions. From axioms VAR12 through VAR16 it follows that an assignment is always treated like *skip*. The axioms for interrupts and disruption are valid only for assignments and closed actions. The axioms for the priority operator hold only for closed actions.
- Standard scope rules apply. The first occurrence of v in *var(v:=s, x)* is the binding occurrence of variable v. The scope of this variable is expression x. The term s is not in its scope, so references to v in s are not bound.
- From the equations above it follows that a reference to a variable which is not initialized yet, leads only to a deadlock if this reference occurs in a normal action, that is, not in an assignment. Such a reference in an assignment or in the initialization part of a var operator is not considered harmful.
- In VAR7 we have the condition that *[w]* does not occur in s to overcome the following problem. Applying this equation without the condition to the expression

  var(w:=0 in D . var(v:=[w] in D, [w:=1] . a([v])))

results in *skip . a(1)* instead of the intended meaning *skip . a(0)*.

4.2.3. Operational Semantics

The transition rules for state variables are defined with the use of an explicit state, which contains all variables and their current values. So a state ϕ is a function from V to the collection of all closed data terms plus the special symbol $\uparrow$, with the requirement that $\phi([v])$ has the same type as v if it is not equal to $\uparrow$.

Now, if P is the class of process expressions and S the class of states, the transition relation $\rightarrow$ is not a relation on P # A # P, but a relation on (S # P) # A # (S # P). We also consider the termination relation $\rightarrow\surd$ on (S # P) # A # S. We will present only the relevant rules concerning state variables. All other rules extend in a straightforward way to the case with states appended.

In the obvious way we extend the domain of the function ϕ to the class of all data terms, with the addition that $\phi(t) = \uparrow$ if t contains $[v]$ such that $\phi([v]) = \uparrow$. An atomic action a is defined in state ϕ, notation $\downarrow(\phi,a)$, if a contains no $[v]$ such that $\phi([v]) = \uparrow$. If $\downarrow(\phi,a)$ holds, we define $\phi(a)$ as the function that applies ϕ to all data terms in a.

SV1	$\dfrac{\downarrow(\phi,a)}{<\phi,a> \xrightarrow{\phi(a)} <\phi,\surd>}$
SV2	$<\phi,[v:=t]> \xrightarrow{skip} <\phi([v]/\phi(t)), \surd>$
SV3	$\dfrac{<\phi([v]/\phi(s)),x> \xrightarrow{a} <\phi',\surd>}{<\phi,var(v:=s \text{ in } D, x)> \xrightarrow{a} <\phi'([v]/\phi([v])),\surd>}$
SV4	$\dfrac{<\phi([v]/\phi(s)),x> \xrightarrow{a} <\phi',x'>}{<\phi,var(v:=s \text{ in } D, x)> \xrightarrow{a} <\phi'([v]/\phi([v])),var(v:=\phi'([v]) \text{ in } D, x')>}$

table 8 Transition rules for state variables

4.2.4. Recursion

Together with recursion, state variables make it possible to use process calls with output parameters. This enables a better modularization, by facilitating the division of processes into subprocesses.

We consider two kinds of indexing of a process: *value* and *reference* indexing. Value indexing is the standard way of indexing in PSF. This means that the index of the defining occurrence of a process (that is: in the left-hand side of a process definition) consists of a data term which may contain static variables but no state variables. A calling occurrence may consist of a data term containing both static and state variables. When the process call is evaluated, the actual values of the state variables in the calling process will be used. The following specification shows an example of value indexing.

P = var(w:=0 in D, b([w]) . X(3, [w], [w]+1) . b([w])).
X(p, q, r) = a(p, q, r),

Process *P* equals the following expression.

b(0) . a(3,0,1) . b(0)

The second way of indexing will be called reference indexing. This is the case if the defining occurrence of a process is indexed with a state variable. The scope of this variable ranges over the entire right-hand side of the defining equation. A calling occurrence may consist only of a state variable.

The intuition is that the formal state variable in the definition is identified with the actual state variable from the calling process. So, if *X* is defined as follows:

X([v]) = a([v]). [v:=v+1] . a([v]),

then a call to *X* with argument *[w]* occurs in the following expression:

var(w:=0 in D, b([w]) . X([w]) . b([w])).

This process equals the following:

b(0) . a(0) . skip . a(1) . b(1)

Note that the state variable *v* in the defintion of process *X* is bound, and thus subject to α-conversion.

Of course we allow both value indexing and reference indexing in one process definition, as long as the definitions are left linear and consistent. This means that a state variable may only occur once in each left-hand side and that all definitions of the same process name and the same type of arguments agree on which indices are value type and which are reference type. So the definitions of processes *X* and *Y* are not allowed in the following example.

X([v],[v]) = ... --not allowed, since not left linear
Y([v]) = ...
Y(0) = ... --not allowed, since the index of Y is not consistent

A call *Z(0)* is not allowed if *Z* is defined by
Z([v]) = ...

On the other hand it is allowed to call *Z([v])* if *Z* is defined by
Z(0) = ...

In this case the actual value of variable v will be matched with 0.

Since an index of a defining occurrence may only be a state variable or a term without state variables, the following definition of *P* is correct, while the definition of *Q* is not.

P(s(x)) = ... --allowed
Q(s([v])) = ... --not allowed

4.2.5. Semantics of Recursion

The algebraic properties of state variables as arguments of a process name deal with the two cases of indexing. The first rule tells how to replace state variables by their value if the variable is in a value index position. The second rule handles substitution of variables in a reference index position. Let *X* be a process name, indexed with both reference and value indices. For ease of notation we assume that the initial indices of *X* are reference indices $[\underline{v}]$ and that the final indices $\underline{t}([\underline{w}])$ are all value indices, that is data terms, possibly containing state variables $[\underline{w}]$. The first state variable in $\underline{t}([\underline{w}])$ is $[w_0]$. Let $\underline{r}$ be a list of data terms without state variables and σ be a substitution of variables. The expression $x \geq y$ is used to denote that *y* is a subprocess of *x*, that is, the equation $x = x + y$ holds.

VARREC1	$X([\underline{v}], \underline{t}([\underline{w}])) = sum(d \text{ in } D, [w_0]=d :\text{->} X([\underline{v}], \underline{t}([w_0]/d)))$
	if d is not in $\underline{t}$
VARREC2	$X([\underline{v}], \underline{r}) \geq \sigma(P([\underline{w}]/[\underline{v}]))$
	if there is a definition $X([\underline{w}], \underline{s}) = P([\underline{w}])$, such that $\sigma(\underline{s}) = \underline{r}$

table 9 Algebraic laws for state variables and recursion

The first rule states that we may substitute all state variables occurring in a value reference by a constant value. This is done from left to right, thus introducing a number of sum-constructs. The second rule states that if we encounter an indexed process name, which matches some process definition, we may conclude that its defintion is a subexpression. We use the ≥-operator instead of equality, since the process might match with more than one definition.

The operational semantics are given in the following table. let *X* be a process name, indexed with a list of state variables $[\underline{v}]$ and a list of (extended) data terms $\underline{t}$. The rules state the following. if *X* is defined by process *y* and process *y* can do an *a* action then *X* can do the same action. We require that the formal state variables $[\underline{w}]$ in *y* are replaced by the actual state variables $[\underline{v}]$, ,Of course variables from $[\underline{v}]$ which occur already in *y* have to be renamed into a list of fresh variables $[\underline{v}']$.

$$\text{R.VREC1} \quad \frac{\langle\phi, y([\underline{v}]/[\underline{v}'])([\underline{w}]/[\underline{v}]) \xrightarrow{a} \langle\phi', y'\rangle;\ X([\underline{w}],\phi(\underline{t})) = y}{\langle\phi, X([\underline{v}],\underline{t})\rangle \xrightarrow{a} \langle\phi', y'\rangle}$$

$$\text{R.VREC2} \quad \frac{\langle\phi, y([\underline{v}]/[\underline{v}'])([\underline{w}]/[\underline{v}]) \xrightarrow{a} \langle\phi', \surd\rangle;\ X([\underline{w}],\phi(\underline{t})) = y}{\langle\phi, X([\underline{v}],\underline{t})\rangle \xrightarrow{a} \langle\phi', \surd\rangle}$$

table 10 Transition rules for state variables and recursion

4.2.6. Input of Data

The feature of state variables can be used to introduce a shorthand notation for the summation construction which is used for reading in data. If the index of an atomic action is a state variable prefixed with a question mark, we interpret this as a summation over all possible values of this variable. So *a(?[d])* is shorthand for *sum(e in D, a(e).[d:=e])*, where *d* is of type *D* and *e* is some new static variable. Now there is a very elegant solution of the parallel input problem from [108]:

```
P = var(x in D, var(y in D, var(z in D,
            (r1(?[x]) ||
             r2(?[y]) ||
             r3(?[z]) ) . s4(x+y+z))))
```

So we have the following semantics of this abbreviation.

$$\text{INP} \quad a(?[v_1], \ldots, ?[v_n]) = \quad \text{sum}(\, w_n \text{ in } D_n, \ldots \text{sum}(\, w_1 \text{ in } D_1, a(w_1, \ldots, w_n).[v_1{:=}w_1]. \ldots . [v_n{:=}w_n])$$

table 11 The input operator

For reasons of clarity the arguments of action *a* not starting with a question mark are left out. The general case follows easily from this definition.

If two input variables are equal, it follows that the value read in for the rightmost occurrence will be the final value of that variable.

4.2.7. Remarks

Note that the use of state variables makes it possible to define (sub)processes with output parameters. This way a process can be split into subprocesses more easily, resulting in better support of information hiding principles.

After having introduced state variables it is also possible to define other imperative constructs like *while* loops. In the following example the process X is defined by repeatedly executing Y, as long as the equation *s*=*t* holds. The data term *b* is a boolean expression, which may contain state variables.

X = **while** b **do** Y **od**

A possible definition which makes use of an additional skip action could be

X = **case** b = true **do** Y . X **od**
= false **do skip od**

Extensions of this kind will not be considered in more detail.

4.2.8. Relation with the State Operator

We can relate the var-operator easily to the state operator (Λ) from [5] in the case that all data terms involved are closed. Every var operator relates to one application of the state operator, with the variable as object and its sort united with $\uparrow$ as domain. The action and effect functions result directly from the axioms above.

In the context of recursion and open data terms, the notion of a state variable provides more than a useful abbreviation scheme for the Λ-operator. Occurrence of a state variable as index of a process name now can be used to see a recursive process call as a procedure call with output parameters.

4.3. An Application: Asynchronous Communication

Communication between processes in PSF is synchronous in the sense that both parties have to take part in the communication at the same instant. In some cases this might not be the desired situation, if for example the sending party only wants to deliver a message, without waiting for synchronisation with the receiver.

Asynchronous communication in ACP was first discussed in [27], where a mechanism was defined that used some newly introduced operators. In [14] the state operator was used for this purpose. In the following section we study the use of state variables to model asynchronous communication between processes.

The basic idea is that messages can be passed asynchronously by using shared state variables. The type of the shared variable indicates which kind of queueing mechanism is used. This leads to a flexible definition of a message queue. Consider for example a regular FIFO-queue to store incoming messages, as defined in the following algebraic specification.

```
data module Queues
begin

  parameters Data
    begin
      sorts
        Data
      functions
        default : -> Data
    end Data

  exports
    begin
      sorts
        Queue
      functions
        empty :                -> Queue
        add   : Data # Queue -> Queue
        enq   : Data # Queue -> Queue    --enqueue at the back
        top   : Queue          -> Data
        pop   : Queue          -> Queue
        empty : Queue          -> BOOL
    end

  imports
    Booleans

  variables
    d,e : -> data
    q   : -> Queue

equations
[1] enq(d, empty) = add(d, empty)
[2] enq(d, add(e, q)) = add(e, enq(d, q))
[3] top(empty) = default
[4] top(add(d,q)) = d
[5] pop(empty) = empty
[6] pop(add(d,q)) = q
[7] empty(empty) = true
[8] empty(add(d,q)) = false

end Queues
```

Using this queue we define a system consisting of a producer and a consumer. The producer reads some data element from its input channel and enqueues this element in the queue. The consumer checks whether the queue is empty and if this is not the case, the top element is processed, that is, sent to the output channel. Although it looks as if the consumer deadlocks when the queue is empty, the overall system can make progress because the producer can do an input action.

```
process module Asynchronous-Communication-with-Queue
begin

  exports
    begin
      processes
        Prod   : Queue
        Cons   : Queue
        System : Queue
      atoms
        input  : Data
        output : Data
    end

  imports
    Queues

  variables
    q : -> Queue

definitions
Prod([q]) = sum(d in Data, input(d) . [q:=enq(d,[q])] . Prod([q]))
Cons([q]) = case empty([q])=false
            do output(top([q])) . [q:=pop([q])] . Cons([q]) od
System = var(q:=empty in Queue, Prod([q]) || Cons([q]))
end Asynchronous-Communication-with-Queue
```

Note that the semantics of the conditional choice and the parallel composition lead to the observation that between testing whether the queue is not empty and sending the top of the queue to the output, there is no possibility for the producer to alter the contents of the queue. Thus we have an implicit locking mechanism which is necessary for correct operation of the system.

In this way any queueing mechanism which can be expressed in an algebraic specification can be used, such as bounded queues, priority queues and stacks. As straightforward as this seems, there is a problem in locking the queue. This occurs if we replace the queue in the previous example by a stack, as defined in the following module.

```
data module Stacks
begin

  parameters Data
    begin
      sorts
        Data
      functions
        default : -> Data
    end Data

  exports
    begin
      sorts
        Stack
      functions
        empty :                -> Stack
        add   : Data # Stack -> Stack
        enq   : Data # Stack -> Stack
        top   : Stack        -> Data
        pop   : Stack        -> Stack
        empty : Stack        -> BOOL
      end

    imports
      Booleans

    variables
      d,e : -> data
      q   : -> Stack

  equations
  [1'] enq(d, q) = add(d, q)
  [3] top(empty) = default
  [4] top(add(d,q)) = d
  [5] pop[(empty) = empty
  [6] pop(add(d,q)) = q
  [7] empty(empty) = true
  [8] empty(add(d,q)) = false

  end Stacks

  process module Asynchronous-Communication-with-Stack
  begin

    exports
      begin
        processes
          Prod   : Stack
          Cons   : Stack
          System : Stack
        atoms
          input  : Data
          output : Data
      end

    imports
      Stacks

    variables
      q : -> Stack

  definitions
  Prod([q]) = sum(d in Data, input(d) . [q:=enq(d,[q])] . Prod([q]))
  Cons([q]) = case empty([q])=false
              do output(top([q])) . [q:=pop([q])] . Cons([q]) od
  System = var(q:=empty in Stack, Prod([q]) || Cons([q]))

  end Asynchronous-Communication-with-Stack
```

Now the producer can add a new element just after the consumer reads the top of the stack and just before the consumer removes it from the stack. An example of an incorrect execution is the following (the skip actions are suffixed with the actions they originate from):

```
input(d) . skip[q=add(d,empty)] . output(d) . input(e) .
skip[q=add(e,add(d,empty))] . skip[q=add(d,empty)] . output(d) .
skip[q=empty]
```

The source of this problem is that reading the top of the stack and removing it are two separate atomic actions, which allow other actions to happen in between. A possible solution to this problem is to give the queue a more complex structure, which we will call a buffer. This buffer consists of the queue itself and the most recently popped data element.

```
data module Buffers
begin

  exports
    begin
      sorts
        Buffer
      functions
        pair  : Stack # Data -> Buffer
        enq   : Data # Buffer -> Buffer
        top   : Buffer -> Data
        pop   : Buffer -> Buffer
        empty : Buffer -> BOOL
    end

  imports
    Stacks

  variables
    d,e : -> data
    q : -> Stack

equations
[1] enq(d, pair(q,e)) = pair(enq(d,q), e)
[2] pop(pair(q,e)) = pair(pop(q),top(q))
[3] top(pair(q,e)) = e
[4] empty(pair(q,e)) = empty(q)

end Buffers
```

The specification of the system now looks as follows:

```
Prod([q]) = sum(d in Data, input(d) . [q:=enq(d,[q])] . Prod([q]))
Cons([q]) = case empty([q])=false
            do [q:=pop([q])] . output(top([q])) . Cons([q]) od
System = var(q:=pair(empty, default) in Buffer,
           Prod([q]) || Cons([q]))
```

One can come up with another solution which consists of a generalization of assignment actions. The idea is based on the observation that the queue is not locked while the consumer is in its critical region. This locking is easily established if we group multiple assignments into one atomic action. In this setting, the following specification of the consumer would suffice.

```
Cons([q]) = case empty([q])=false
              do var(temp in Data,
                    [temp := top([q]); q:= pop([q])] .
                    output([temp])) . Cons([q])
              od
```

Here the semicolon is used to separate the assignments and the intended semantics is the obvious semantics.

It is worthwile to note that this treatment of asynchronous communication does not reflect the situation in most object-oriented programming languages where every object claims a queue as its own property. In the setting described above, the producer, and, in fact, any other process that shares the queueing variable, has the possibility to alter the contents of the queue other than by means of enqueing a message. Exactly the same problem holds for the setting with synchronous communication, where a communication channel does not belong to a pair of two processes that want to communicate along that channel, but to any process willing to write or read on the channel. Still a setting where each object has a clear identification and a private message queue might be desirable.

5. An Example

In the following example we will use the new features introduced in this chapter. This example shows the operation of a television control. The behaviour was reconstructed using reverse engineering on an existing tv set.

We start with a specification of the Booleans with values *true* and *false*.

```
data module Booleans
begin

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

end Booleans
```

The state of some properties can be *on* or *off*. States are toggled by applying the *not* function.

```
data module Status
begin

  exports
    begin
      sorts
        Status
        functions
          on  :        -> Status
          off :        -> Status
          not : Status -> Status
      end

  equations
  [1] not(on) = off
  [2] not(off) = on

  end Status
```

Volume and brightness can have a value in a subrange of the naturals. The minimum value is 0 and the maximum value is 20. The optimal adjustment *med* is at 10. Within its range, a value can be incremented and decremented.

```
  data module Values
  begin

    exports
      begin
        sorts
          Val
        functions
          0   :     -> Val
          s   : Val -> Val
          inc : Val -> Val
          dec : Val -> Val
          max :     -> Val
          med :     -> Val
      end

    imports
     Booleans

    functions
     lt : Val # Val -> BOOL

    variables
     x, y : -> Val

  equations
  [1]  max = s(s(s(s(s(s(s(s(s(s(
                s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))))    --20
  [2]  med = s(s(s(s(s(s(s(s(s(s(0))))))))))                  --10
  [3]  inc(x) = s(x)    when lt(x, max) = true
  [4]  inc(x) = x       when lt(x, max) = false
  [5]  dec(s(x)) = x
  [6]  dec(0) = 0
  [7]  lt(0, s(y)) = true
  [8]  lt(x, 0) = false
  [9]  lt(s(x), s(y)) = lt(x, y)

  end Values
```

There are a number of keys to control volume, brightness, channel and teletext. The quiet key toggles between no volume and the current volume, the optimal key resets all values, toggle-tt switches teletext on and off. The keys 0 up to 9 are used to select a channel. The number of the selected channel is of type *Val* and can be computed by the function *val*.

```
data module Keys
begin

  exports
    begin
      sorts
        Key
      functions
        up-vol    : -> Key
        dwn-vol   : -> Key
        up-bri    : -> Key
        dwn-bri   : -> Key
        quiet     : -> Key
        optimal   : -> Key
        toggle-tt: -> Key
        suspend   : -> Key
        0-key     : -> Key
        1-key     : -> Key
        2-key     : -> Key
        3-key     : -> Key
        4-key     : -> Key
        5-key     : -> Key
        6-key     : -> Key
        7-key     : -> Key
        8-key     : -> Key
        9-key     : -> Key

        val : Key -> Val
    end

  imports
    Values

equations
[1]  val(up-vol)          = 0
[2]  val(dwn-vol)         = 0
[3]  val(up-bri)          = 0
[4]  val(dwn-bri)         = 0
[5]  val(quiet)           = 0
[6]  val(optimal)         = 0
[7]  val(toggle-tt)       = 0
[8]  val(suspend)         = 0
[9]  val(0-key)           = 0
[10] val(1-key)           = s(0)
[11] val(2-key)           = s(s(0))
[12] val(3-key)           = s(s(s(0)))
[13] val(4-key)           = s(s(s(s(0))))
[14] val(5-key)           = s(s(s(s(s(0)))))
[15] val(6-key)           = s(s(s(s(s(s(0))))))
[16] val(7-key)           = s(s(s(s(s(s(s(0)))))))
[17] val(8-key)           = s(s(s(s(s(s(s(s(0))))))))
[18] val(9-key)           = s(s(s(s(s(s(s(s(s(0)))))))))

end Keys
```

A television set consists of a number of components. These components are not specified, but their names are used for addressing messages. We assume controls for powering down (parts of) the television set, for sound, display, teletext and channel selection. Furthermore there is a small light to give user feedback and a memory to store status information when the television is switched off.

```
data module Components
begin

  exports
    begin
      sorts
        Component
      functions
        power-control   : -> Component
        sound-control   : -> Component
        display-control : -> Component
        tt-control      : -> Component
        chan-control    : -> Component
        led             : -> Component
        mem             : -> Component
    end

end Components
```

Messages that are sent to the various components consist of a value or a status. The LED can receive a *flash* message.

```
data module Messages
begin

  exports
    begin
      sorts
        Message
      functions
        msg    : Val    -> Message
        msg    : Status -> Message
        flash  :        -> Message
    end

   imports
    Values, Status

end Messages
```

The process *TV-Control* starts with a *power-on* action, followed by reading in the stored values of channel, volume and brightness. Then it continues with the process *Control*, initialized with these values, while *ttstat*, *suspend* and *qstat* are *off* and the *oldvolume* is 0. This means that teletext is not active, the television is not suspended and that the sound is not switched off.

Operation of the *TV-Control* can be disrupted by a *power-off* action at any instant. Before actually quitting, there is some time left to store the current values of *channel, volume* and *brightness* in memory for later use.

The *Control* process starts by receiving a key press. Then, depending on whether the television is suspended or not, this key is interpreted. After suspension, operation can only be resumed if a number key is pressed. This then becomes the active channel.

If operation is not suspended, the keys have the following result. Pressing the *suspend* key forces suspension. The volume and brightness controls determine the values of *vol* and *bri*. Sound can be turned off temporarily by pressing the *quiet* key. Pressing it a second time restores the old volume. Standard values for volume and brightness can be set using the *optimal* key. Toggling teletext on and off is done with the *toggle-tt* key. If teletext is on, the keys 0 up to 9 control the displayed page. Otherwise they control the channel.

```
process module TV-Control
begin

  exports
    begin
      atoms
        r : Component # Message
        s : Component # Message
        receive : Key
        power-on, power-off
      processes
        TV-Control
        Control : Val # Val # Val # Status # Status # Status # Val
    end

  imports
    Status, Keys, Messages, Components

   variables
    chn, vol, bri, col, oldvol : -> Val
    ttstat, susp, qstat : -> Status

definitions
  TV-Control =
    var(chn in Val, var(vol in Val, var(bri in Val,
    var(ttstat:=off in Status, var(susp:=off in Status,
    var(qstat:=off in Status, var(oldvol:=0 in Val,
      power-on .
      dis(r(mem, msg(?[chn])) .
            r(mem, msg(?[vol])) .
            r(mem, msg(?[bri])) .
            Control([chn], [vol], [bri], [ttstat],
                                  [susp], [qstat], [oldvol]),
          power-off .
            s(mem, msg([chn])) .
            s(mem, msg([vol])) .
            s(mem, msg([bri])))))))))
```

```
Control([chn],[vol],[bri],[ttstat],[susp],[qstat],[oldvol]) =
  var(k in Key,
   receive(?[k]) .
   case
    [susp] = off
     do
      s(led, flash) .
      case [k]
        = suspend    do [susp:=on] .
                        s(power-control, msg([susp])) od
        = up-vol     do [vol:=inc([vol])] .
                        s(sound-control, msg([vol])) od
        = dwn-vol    do [vol:=dec([vol])] .
                        s(sound-control, msg([vol])) od
        = up-bri     do [bri:=inc([bri])] .
                        s(display-control, msg([bri])) od
        = dwn-bri    do [bri:=dec([bri])] .
                        s(display-control, msg([bri])) od
        = quiet      do case qstat
                          = off do [oldvol:=[vol]] .
                                   [vol:=0].[qstat:=on] od
                          = on  do [vol:=[oldvol]] .
                                   [qstat:=off] od .
                        s(sound-control, msg([vol])) od
        = optimal    do [vol:=med] .
                        s(sound-control, msg([vol])) .
                        [bri:=med] .
                        s(display-control, msg([bri])) od
        = toggle-tt  do [ttstat:=not([ttstat])] .
                        s(tt-control, msg([ttstat])) od
        = 0,1,2,3,4,5,6,7,8,9
                     do case ttstat
                         = on
                           do s(tt-control, msg(val([k]))) od
                         = off
                           do [chn:=val([k])] .
                              s(sound-control, msg(0)) .
                              s(chan-control, msg([chn])) .
                              s(sound-control, msg([vol])) od
                     od .
      Control([chn], [vol], [bri], [ttstat],
                                  [susp],[qstat],[oldvol])
    = on
     do case k
          = 0,1,2,3,4,5,6,7,8,9
             do s(led, flash) . [susp:=off] .
                [ttstat:=off] . [oldvol:=0] .
                s(power-control, msg([susp])) .
                [chn:=val([k])] . s(chan-control, msg([chn])) .
                s(sound-control, msg([vol])) .
                s(display-control, msg([bri])) .
                Control([chn], [vol], [bri], [ttstat],
                                  [susp], [qstat], [oldvol]) od
          = suspend, up-vol, dwn-vol, up-bri,
                  dwn-bri, quiet, optimal, toggle-tt
             do Control([chn], [vol], [bri], [ttstat],
                                  [susp], [qstat], [oldvol]) od
     od
end TV-Control
```

6. CONCLUSION

The orientation of PSF on ACP and the fact that PSF is meant to be a computer manageable formalism seem to be in contradiction. New operators are continuously added to ACP, in order to increase the expressiveness, to obtain more concise specifications or to make verification easier. In contradiction with this, a computer formalism should remain stable and new releases of such a language should be scarce and backwardly compatible.

The best way to manoeuvre through this contradiction is to find a compromise between stability and extendability. New features should only be added if they have proven to be very useful, or even to be necessary, for specification or verification purposes. This is to be established in case studies.

A requirement for such extensions will also be that the impact on the specification language will be as small as possible. Thus the addition of a new operator should result only in the addition of new syntax for this operator. Addition of a new process operator will only influence the definition of process operators, without altering the modularization concept or the way in which data types are specified. This requirement sometimes makes it necessary to make a redesign of the operators involved, instead of simply copying them. This could be the case for example if a newly introduced process operator makes assumptions on the structure of the data types, such as the existence of predefined functions or sorts.

As a last requirement it is stated that newly introduced process operators have a semantics which can be defined using transition rules, in such a way that these rules can be composed with the existing rules without altering previously defined operators.

Now let us have a look at the three candidates for extending PSF which are considered in this chapter. Conditional choices seem to be very useful for making the *control structure* of a process execution more visible. It helps to avoid unnecessary process definitions, which are only added to sum up all choice alternatives. As a side-effect, the use of conditional choices makes it possible to consider the subset of PSF which only allows variables as arguments of the process names at the left hand side of a process definition. The advantage of this subset is that the equality sign in a process definition does not have to be interpreted as a summand sign, which resolves some intuitive as well as some computational problems related to this interpretation.

The second extension studied consists of three operators, which together enable the specification of interruptable systems. These are the interrupt and disabling operator, which add the possibility of a process to be interrupted or disabled at any time, and the modified priority operator which assures that an interrupting action occurs whenever possible. Instead of introducing the notion of a partial order on atoms (which is used in the priority operator as defined for ACP) we modify this operator and use the already existing notion of sets.

The third example, state variables, is a revision of the state operator, for which the definition of an action and effect function is assumed. State variables behave as variables in a regular programming language, thus introducing imperative aspects in the PSF language. They are useful for specifying processes with parallel input, processes with output parameters and to keep track of the state of a process.

We may conclude that without major difficulties many extensions of ACP can be added to PSF at the cost of a simple redesign of the operators involved. The question which features must be added in a new release of PSF is not answered yet. Other extensions which are candidate to be part of PSF are real time process algebra, probabilistic choices and process creation.

To overcome the problems with extending PSF, it might be argued that a real ACP based specification language should not fix the set of admissible process operators. Such a language, and of course all its tools, should have this set as a parameter. Thus the user will have to specify for each use the operators to work with, their syntax, the axioms which they obey and the transition rules defining their behaviour. This idea is implemented in the Process Algebra Manipulator [77].

A drawback of this approach is that tools can only use the information provided through this parameter, which can have considerable consequences for the speed of execution of the tools. In the current implementation for PSF heuristics are used which for example handle combinations of several operators.

For the moment we restrict ourselves to a language with a predefined set of process operators.

Chapter 4

A Tool Interface Language for PSF

(with G.J. Veltink)

Syntax and semantics of a Tool Interface Language (named TIL) for PSF (a Process Specification Language) are defined. TIL is meant to be an intermediate language between the various tools under development for PSF, such as tools for simulation, verification and implementation.

1. Introduction

When creating a programming environment for a specification language for concurrent systems, there are several reasons not to let the interaction between the various tools take place at the specification language level. An interface language can be used to make a layered design, such that tools act on a low level, while humans can inspect a high level representation. If the structure of such high level language does not allow for easy parsing and type checking, a translation to a language that is simpler to parse could be beneficial for the complexity of the tools. The effort of writing a complex parser and type checker has to be done only once, while the other tools only need a simple parser to read the intermediate language.

If the high level language is a member of a set of similar languages with comparable functionality, the toolkit can easily be adapted for another member of this group by only writing a new front-end to the intermediate language.

These are the reasons for choosing to use a Tool Interface Language (TIL) for the toolkit under development for the PSF language.

This approach is similar to the one taken in the RACE project SPECS. There CRL (Common Representation Language) [98] is used as the greatest common divisor of several specification languages. Though TIL and CRL differ in the sense that they both have features that cannot be expressed in the other language, the main difference is a difference in style. We expect that a translation from one language into the other will be easy, when (and if) the missing features are added.

As in CRL, TIL features the notion of a *hook*, in order to make links to the high level representation of the specification. This is done using a so called *free format* field.

The concept of a tool interface language is not new. The standard Ada programming support environment is often built around the Diana language [49], which is an attributed notation for Ada programs. This intermediate language also allows the tools to store local information and helps in reducing the overhead in parsing.

In this chapter we present syntax and semantics of TIL. A specification in TIL consists of a number of tuples, each declaring or defining one item. The semantics of TIL are defined using initial algebra semantics and action relations. An example is included, showing a PSF specification and the corresponding TIL text. A first proposal for the language TIL is in [68], on which parts of this chapter are based.

1.1. DESIGNING TIL

A Tool Interface Language for PSF has to meet several criteria. First of all from a semantical point of view it must have the expressiveness of process algebra (ACP) and algebraic specifications. This will make it possible to give a mapping from PSF specifications to TIL specifications having the same meaning. Since the way a PSF specification is split up into modules is immaterial for the semantics, TIL should not cover these modularization features. Other features from PSF that are not supported by TIL are: overloading of names, renaming, parameterization, user-defined operators and tuples of terms over data types.

In contrast to PSF, which should be easily readable for humans, TIL should be easily accessible by computer tools. Parsing TIL should be easy, while readability is of minor importance. In many respects TIL can be compared to an assembler language.

Because TIL is an interface between several computer tools, not known in advance, it must have a mechanism to allow the tools to insert information of a type and in a format that is not dictated by TIL. Such free format information is dependent on the tools themselves.

Altough TIL is used as an interface language for PSF tools, TIL is meant to be a language not dependent on PSF. It is defined in such a way that simple extensions will make it suitable for other high level specification languages, such as LOTOS [67] and μ-CRL [51].

2. SYNTAX OF TIL

In the following sections the syntax of the Tool Interface Language will be described. We also explain the use of free formats.

2.1. GENERAL

A specification in TIL consists of a series of tuples. The order in which the tuples appear is immaterial. A tuple can be viewed as the declaration or definition of one item, for example a sort name, an equation or a process definition. (These tuples should not be confused with the data structuring mechanism of the same name in ASF.)

2.2. TUPLE LAYOUT

For each kind of item we have defined a tuple layout. In general a tuple looks like:

key definition free-format

The *key* is the name by which we can refer to the item defined by its *definition* throughout the entire specification. The *definition* field may sometimes be empty. All keys should be unique in the entire specification. In the *free-format* section information concerning the defined item can be recorded, which can be used to exchange information between the tools. The contents of the *free-format* fields are disregarded when determining the semantics of the specification. We will elaborate on the use of the free formats later.

The general form of a key is the following:

[X.Y]

where X identifies the type of item, and Y is an identifier, used to generate unique keys. The value of Y must be a natural number. The value of X can be one of the following:

0	administration
1	sort declaration
2	function declaration
3	atomic action declaration
4	process declaration
5	set declaration and definition
6	communication definition
7	variable declaration
8	equational specification
9	process specification

figure 2.1 Types of a tuple

The meaning of these items and their format are defined below.

Operators (on sets or processes) and predefined processes are denoted by *<id,#arg>* or *<id>*. Here *id* is an identifier, determining the operation, and *#arg* is a natural number, denoting the number of arguments. The following operators are defined.

<:,n>	enumeration of set elements
<+,n>	union of sets
<.,n>	intersection of sets
<\,n>	difference of sets (left associative)
<alt,n>	alternative composition of processes (+)
<seq,n>	sequential composition of processes (.)
<par,n>	parallel composition of processes (||)
<sum>	generalized alternative composition (sum)
<merge>	generalized parallel composition (merge)
<encaps>	encapsulation of atomic actions (encaps)
<hide>	hiding of atomic actions (hide)
<skip>	internal action (pre-abstraction)
<delta>	deadlock action (delta)
<if>	conditional expression
<case>	multiple conditional expression

figure 2.2 Operators on sets and processes.

There is no precedence defined on the operators. If brackets are omitted, expressions are associated from left to right.

In the following examples we already include free format fields. These fields give an example of their use but are not essential for the tuple definition.

Comments A comment starts with a double-hyphen (--) and ends with an end-of-line. Comments may contain any character but the end-of-line character.

0 **Administration** This tuple can be used to store tool dependent information about the specification. See the section on free formats for a more thorough treatment of this kind of information.

```
        [0.1] {<date> 19890811}        --This is a comment
```

This defines an administration tuple with key [0.1], expressing the fact that the specification was created on a certain date.

1 **Sort declaration** Declaration of a sort, an abstract data type.

```
        [1.1] {<n> Bool}
```

This tuple declares a sort which can be referred to by [1.1]. In the free format section the intended name of the sort is added: *Bool*. This is indicated by the *<n>* expression.

2 **Function declaration** A function declaration contains the type of the arguments and the result. The input type may consist of zero or more sort references, while the output type consists of one sort. The arity (number of arguments) is also part of the declaration.

[2.1] 2 [1.1] [1.1] [1.1] {<n> and}

The boolean function *and* is declared, having two booleans as input and one boolean as output.

3 **Atomic action declaration** An atomic action is treated similarly as a function, but only has an input type.

[3.1] 2 [1.2] [1.1] {<n> read}

The *read* action has two arguments, the first of which is a sort identified by [1.2] (a channel for example). The second one is a *boolean*.

4 **Process declaration** A process declaration has the same format as the declaration of an atomic action.

[4.1] 0 {<n> Start}

The process *Start* has no arguments.

5 **Set declaration and definition** Sets are used to define a subset of some previously defined sort. Each set has a sort assigned to it. This sort is given in the first part of the definition. The second part determines the members of the set. Sets can be constructed by enumeration and by applying set operators (union, intersection, difference).

[5.1] [1.3] <:,2> ([2.16] [2.17]) {<n> weekend}

Although for parsing the parentheses in the example are redundant, in TIL they are demanded for reasons of readability.

Given a sort [1.3] (days of the week) and constant functions of this sort [2.11] - [2.17] (*monday* through *sunday*), we define the set [5.1] (*weekend*) containing *saturday* and *sunday* only.

[5.2] [1.3] <\,2> ([1.3] [5.1]) {<n> working-days}

This defines the complement of *weekend*.

When enumerating terms, also terms containing variables (ranging over a sort or a set) may be used. This means that all terms resulting from substituting closed terms for these variables are member of the set.

[5.3] [1.4] <:,1> ([2.2] ([7.1])) {<n> even-naturals}

Let [1.4] denote the sort *integer*, [2.2] denote the function *double*, and let [7.1] be a variable over sort *integer*. Now [5.3] defines the set of *even-naturals*. Note that variables are implicitely bound by the <:,n> operator.

To be able to create sets of atomic actions, the special "sort" [1.0] can be used as the sort associated to such a set.

[5.4] [1.0] <:,3> ([3.2] [3.3] [3.4]) {<n> internal-actions}

This defines the set [5.4], containing the actions [3.2] [3.3] [3.4].

6 **Communication definition** These tuples define the so called communication function. When two actions are performed in parallel, they can communicate, which results in a new atomic action. The atomic actions may contain variables.

[6.1] [3.3] [3.4] [3.5] {<n> C1}

The result from communication between actions [3.3] and [3.4] is action [3.5].

7 **Variable declaration** Variables can be used in tuples of type 5, 6, 8 and 9. A variable ranges over a sort or a set.

[7.1] [1.4] {<n> m}

A variable of sort [1.4] (*integers*) with name *m* is declared.

8 **Equational specification** In this kind of tuple data types can be defined using conditional equations. Equations consist of a pair of terms of the same sort, possibly succeeded by a list of conditions. Variables occurring in equations are universally quantified.

The following example shows a line from a PSF specification and its equivalent in TIL.

```
[B1]  and(false,b) = false
[8.1] [2.1] ([2.9] [7.2]) = [2.9]   {<n> B1}
```

Conditions are specified using the <= symbol, followed by the number of conditions.

```
[B6]  and(b,c) = false     when b=true, c=false
[8.2] [2.1] ([7.2] [7.3]) = [2.9]   <= 2   [7.2] = [2.8], [7.3] = [2.9]   {<n> B6}
```

9 **Process specification** A process specification consists of two parts. First the process name, having an appropriate number of terms as arguments. Free variables in these terms are universally quantified. The second part is the definition of the process, consisting of a process term without unbound variables.

The following example shows a line from a PSF specification and its equivalent in TIL.

```
X(b) = (skip.send(b) + skip.error) . sum(c in Bool, read(c).X(c))
[9.1] [4.2] ([7.2]) = <seq,2> (<alt,2> (
                  <seq,2> (<skip> [3.6] ([7.2])) <seq,2> (<skip> [3.9])
                  <sum> ([7.3] <seq,2> ([3.8] ([7.3])) [4.2] ([7.3])))) {}
```

2.3. CONTEXT-FREE SYNTAX

In this section we will give the definition of TIL in SDF (Syntax Definition Formalism) (see [56]). 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 LEX [76] and YACC [70].

```
module TIL
exports
  sorts
    Free-Format Free-Format-Char Comment-Char Digit Natural
    Id-First-Char Id-Char Id

  lexical syntax
    [ \n\t]                                      -> LAYOUT
    ~[\n]                                        -> Comment-Char
    "--" Comment-Char* "\n"                      -> LAYOUT
    ~[}]                                         -> Free-Format-Char
    "{" Free-Format-Char* "}"                    -> Free-Format
  [0-9]                                      -> Digit
  Digit+                                     -> Natural
  [0-9a-zA-Z]                                -> Id-First-Char
  [0-9a-zA-Z'\-]                             -> Id-Char
  Id-First-Char Id-Char*                     -> Id

sorts
  Specification Entry Administration-Entry Sort-Entry
  Function-Entry Atom-Entry Process-Entry Set-Entry
  Communication-Entry Variable-Entry Equation-Entry
  Definition-Entry Administration-Index Sort-Index
  Function-Index Atom-Index Process-Index Set-Index
  Communication-Index Variable-Index Equation-Index
  Definition-Index Variable-Type Set-Expr Enumeration-Item
  Atom-Term Term Equation Equation-Expr Definition-Expr
  Process-Head Case-Pair Process-Expr

context-free syntax
  "specification" Id Entry* "end"            -> 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

  Administration-Index Free-Format           -> Administration-Entry
  Sort-Index Free-Format                     -> Sort-Entry
  Function-Index Natural Sort-Index* Free-Format
                                             -> Function-Entry
  Atom-Index Natural Sort-Index* Free-Format  -> Atom-Entry
  Process-Index Natural Sort-Index* Free-Format
                                             -> Process-Entry
  Set-Index Sort-Index Set-Expr Free-Format  -> Set-Entry
  Communication-Index
  Atom-Term Atom-Term Atom-Term Free-Format  -> Communication-Entry
  Variable-Index Variable-Type Free-Format   -> Variable-Entry
  Equation-Index Equation-Expr Free-Format   -> Equation-Entry
  Definition-Index Definition-Expr Free-Format
                                             -> Definition-Entry

  "[0." Natural "]"                          -> Administration-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

  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
Equation                                            -> Equation-Expr
Equation "<=" Natural { Equation ","}+              -> Equation-Expr

Process-Head "=" Process-Expr                       -> Definition-Expr

Process-Index                                       -> Process-Head
Process-Index "(" Term+ ")"                         -> Process-Head

"=" Term Process-Expr                               -> Case-Pair

Process-Head                                        -> Process-Expr
Atom-Term                                           -> Process-Expr
"<skip>"                                            -> Process-Expr
"<delta>"                                           -> 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
"<alt," Natural ">(" Process-Expr+ ")"              -> Process-Expr
"<seq," Natural ">(" Process-Expr+ ")"              -> Process-Expr
"<par," Natural ">(" Process-Expr+ ")"              -> Process-Expr
"<if>(" Term Case-Pair ")"                          -> Process-Expr
"<case," Natural ">(" Term Case-Pair+ ")"           -> Process-Expr
```

2.4. Context-sensitive Syntax

A syntactically correct TIL text has to meet some extra conditions, in order to have a semantical meaning.

- Distinct tuples must have different keys.
- All objects occurring in a specification must be declared and must have correctly typed arguments (if they have any).
- The communication function defined in the communications section should be associative and commutative.The atomic actions *<skip>* and *<delta>* cannot take part in any communication action.
- The part of an operator indicating the number of arguments must match the actual number of arguments.

- Variables over sets may only occur in a set definition (5), a communication definition (6) or in a generalized sum or merge in a process specification (9).
- The left-hand side and the right-hand side of an equation must have the same type.
- There should be exactly one binding occurrence of a variable in a process specification (9). All other occurrences of this variable in that process specification must be within the scope of the binding occurrence. The scope of a variable is defined as in PSF in a straightforward way.
- Operators of the form <*id*,#arg> must have at least two arguments, except for <:,#arg> and <*case*,#arg>, which must have at least one argument.
- Set definitions may not contain cyclic references to the set being defined.
- The second number in a key may be any natural number but zero, except in the definition of a set where the key [1.0] is used to indicate that the set being defined consists of *atomic actions*.

2.5. FREE FORMATS

A tuple in a TIL specification may contain a so called *free format* field. The contents of a free format field are not subject to syntactical rules defined in this chapter. We give however a suggestion of how the free format field could be structured and show how we intend to store information about names and origin of objects. (Origins are introduced in [20]).

The use of the free format field is to allow the exchange of specific information between the various tools acting on a TIL specification. Take as an example the information that a specification defines a regular process, which might be the outcome of a process classification tool. This information can be used by a verification tool to select a proper algorithm.

Other examples are the fact that the data type specification determines a confluent term rewriting system or that a data type is defined using a certain set of constructor functions.

In order to avoid interference of the various kinds of information added by several tools, we propose some structure on the free format field. We adopt the convention that the parts of information are preceeded by a tag indicating the type of information. A tag consists of an identifier enclosed by angle brackets.

The following example shows a free format used to indicate that the processes defined in the specification are regular.

[0.1] {<class> regular}

The tools acting on a TIL specification all may know of a set of tags, indicating information relevant to this tool. Of course these sets may overlap if two tools want to exchange information. The documentation of a tool should indicate which tags are used for what kind of information. Special care should be taken that in the complete tool environment all tags have an unambiguous meaning.

The free format field in a tuple defining an item can be used to store information about that special item. General information about the specification can be stored in a *free format* field of an *administration* tuple.

The following tuple can be inserted by a typechecking tool that succeeded in checking the typing information in the specification.

[0.2] {<type-check> ok}

As a more complicated example we will demonstrate the use of free formats to construct so called *hooks*. A hook is a way to refer to the original specification from which the TIL specification is derived. This is a way to make the intermediate TIL level transparant to a regular user of PSF and PSF-tools. Information generated by a tool acting on TIL in general uses the names of the items as defined in TIL. These names however do not correspond with the names in the original specification, so the original names should be remembered when translating PSF to TIL. The names of the items as they are known in the outside world are stored in the free formats using a name-tag <n>. The following PSF fragment

```
data module junk
begin
  exports
    begin
      sorts
        foo
      functions
        bar : foo -> foo
    end
end junk
```

results in

```
[1.1] {<n>foo}
[2.1] 1 [1.1] [1.1] {<n>bar}
```

Since the name does not uniquely identify an object in PSF (think of hidden objects), we also need information about the *origin* of an object. This origin consists of (a reference to) the name of a module, using an origin-tag <o>. The names of modules can be stored using module-tags <m>. If we add origin information, the previous example expands to:

```
[0.1] {<m>1 junk}
[1.1] {<n>foo <o>1}
[2.1] 1 [1.1] [1.1] {<n>bar <o>1]
```

In case the top-level name of the object is not the same as the name of that object in its origin module we can also add the original name, using the <on> tag. This situation occurs when applying a renaming or when binding a parameter.

```
data module top
begin
  imports junk
    {renamed by
      [ foo -> A,
        bar -> b ] }
end top

[0.1] {<m>1 junk}
[0.2] {<m>2 top}
[1.1] {<n>A <o>1 <on>foo}
[2.1] 1 [1.1] [1.1] {<n>b <o>1 <on>bar]
```

Note that we also included the possibility to add comments to a specification. These comments are different from free formats, since they are intended to contain information for a human reader. Just as free formats, they have no semantical meaning.

3. SEMANTICS

The semantics for TIL are constructed from the semantics of PSF. Note that there is a canonical mapping of TIL specifications into PSF. Every TIL specification corresponds to a flat PSF specification, that is a specification consisting of exactly one data module and one process module. The semantics of the corresponding PSF specification is equal to the semantics of the TIL specification.

See the appendix for a list of action rules for TIL.

4. AN EXAMPLE

In this section we give an example of a PSF specification and the corresponding TIL specification. In the PSF text we used almost all modular constructs in PSF which are not in TIL. The TIL specification could only be constructed after normalizing the original specification, that is, flattening the specification until only one data module and one process module remains.

4.1. RUSSIAN ROULETTE IN PSF

We specify a deadly game in PSF. In the module *Booleans* the constants *true* and *false* and the functions *and* and *or* are defined. The next module contains the *Naturals*, including an equality function on the *Naturals*. The module *Russian-Roulette* has a parameter *Counter*, in which the sort *COUNT* and the function *tick* are presumed. The process *roulette* has one argument of sort *COUNT* and behaves as follows. It has the choice of performing a *skip* action followed by a *click* (no bullet) and a restart of the game having the next chamber active, or it can choose to do a *skip*, followed by a *bang* (indeed a bullet) leading to extermination. Since both possibilities start with a *skip* action, the choice is non-deterministic.

In the next module we make a game of the *roulette* process. This is done by binding the parameter of *Russian-Roulette* to the *Naturals*, renaming some items and placing a process in parallel that keeps the score as the number of tries before extermination. This number is the argument of the *exterminate* action which is renamed to *game-over*. If placed in parallel, the *game-over* action and the *wait* action can communicate, which results in a *result* action. This way the *score* is communicated. In order to avoid unsuccessful communications, all actions from set *H* are encapsulated. This way they can only occur in a communication. Since we are only interested in the result of the game, all internal actions are hidden.

```
data module Booleans
begin

exports
  begin
    sorts
      BOOL
    functions
      true  :               -> BOOL
      false:              -> BOOL
      and  : BOOL # BOOL -> BOOL
      or   : BOOL # BOOL -> BOOL
  end

functions
  not  : BOOL -> BOOL
variables
  x, y   : -> BOOL

equations

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

end Booleans


data module Naturals
begin

exports
  begin
    sorts
      NAT
    functions
      zero  :            -> NAT
      succ  : NAT        -> NAT
      equal : NAT # NAT  -> BOOL
  end
```

```
imports
   Booleans

variables
   x, y   : -> NAT

equations

[N1] equal(zero,zero)         = true
[N2] equal(zero,succ(y))      = false
[N3] equal(succ(x),zero)      = false
[N4] equal(succ(x),succ(y))   = equal(x,y)

end Naturals


process module Russian-Roulette
begin

parameters
   Counter
     begin
       sorts
         COUNT
       functions
         tick   : COUNT   -> COUNT
     end Counter

exports
   begin
     atoms
       exterminate : COUNT
     processes
       roulette : COUNT
     sets of atoms
       Internals = { click, bang }
   end

atoms
   click, bang

variables
   c : -> COUNT

definitions

   roulette(c) = skip . click . roulette(tick(c)) +
                 skip . bang . exterminate(c)

end Russian-Roulette
```

```
process module Game
begin

   imports
     Russian-Roulette
     { Counter
       bound by
         [ COUNT   -> NAT,
           tick  -> succ ]
       to Naturals
       renamed by
         [ exterminate -> game-over,
           roulette    -> game ] }

   atoms
     wait, result : NAT

   processes
     play

   sets of atoms
     H = { game-over(score)  | score in NAT}
       + { wait(score)  | score in NAT }

   communications
     game-over(score)  | wait(score)  = result(score) for score in NAT

   definitions
     play = hide(Internals, encaps(H,
              game(zero)  || sum(score in NAT, wait(score))))

end Game
```

4.2. Russian Roulette in TIL

This time the specification is written down in TIL. The items are grouped by sort, but the order is immaterial. In the free formats references are added to the original PSF specification.

```
[0.1]                                                    { <m>1 Booleans }
[0.2]                                                    { <m>2 Naturals }
[0.3]                                             { <m>3 Russian-Roulette}
[0.4]                                                        { <m>4 Game }
[1.1]                                                        <n>BOOL <o>1}
[1.2]                                                       { <n>NAT <o>2}
[2.1] 0 [1.1]                                              { <n>true <o>1}
[2.2] 0 [1.1]                                             { <n>false <o>1}
[2.3] 2 [1.1]  [1.1]  [1.1]                                 { <n>and <o>1}
[2.4] 2 [1.1]  [1.1]  [1.1]                                  { <n>or <o>1}
[2.5] 1 [1.1]  [1.1]                                        { <n>not <o>1}
[2.6] 0 [1.2]                                              { <n>zero <o>2}
[2.7] 1 [1.2]  [1.2]                                       { <n>succ <o>2}
[2.8] 2 [1.2]  [1.2]  [1.1]                               { <n>equal <o>2}
[3.1] 1 [1.2]                     { <n>game-over <on>exterminate <o>3}
[3.2] 0                                                   { <n>click <o>3}
[3.3] 0                                                    { <n>bang <o>3}
```

```
[3.4] 1 [1.2]                                          { <n>wait <o>4}
[3.5] 1 [1.2]                                        { <n>result <o>4}
[4.1] 1 [1.2]                             { <n>game <on>roulette <o>3}
[4.2] 0                                                { <n>play <o>4}
[5.1]   [1.0] <:,2>([3.2] [3.3])                  { <n>Internals <o>3}
[5.2]   [1.0] <+,2>(<:,1>([3.1]([7.6])) <:,1>([3.4]([7.7])))
                                                          { <n>H <o>4}
[6.1]   [3.1]([7.8]) [3.4]([7.8]) [3.5]([7.8])                 { <o>4}
[7.1]   [1.1]                                             { <n>x <o>1}
[7.2]   [1.1]                                             { <n>y <o>1}
[7.3]   [1.2]                                             { <n>x <o>2}
[7.4]   [1.2]                                             { <n>y <o>2}
[7.5]   [1.2]                                             { <n>c <o>3}
[7.6]   [1.2]                                         { <n>score <o>4}
[7.7]   [1.2]                                         { <n>score <o>4}
[7.8]   [1.2]                                         { <n>score <o>4}
[7.9]   [1.2]                                         { <n>score <o>4}
[8.1]   [2.5]([2.1]) = [2.2]                           { <n>[B1] <o>1}
[8.2]   [2.5]([2.2]) = [2.1]                           { <n>[B2] <o>1}
[8.3]   [2.3]([7.1] [2.1]) = [7.1]                     { <n>[B3] <o>1}
[8.4]   [2.3]([7.1] [2.2]) = [2.2]                     { <n>[B4] <o>1}
[8.5]   [2.4]([7.1] [7.2]) = [2.5]([2.3]([2.5]([7.1]) [2.5]([7.2])))
                                                       { <n>[B5] <o>1}
[8.6]   [2.8]([2.6] [2.6]) = [2.1]                     { <n>[N1] <o>2}
[8.7]   [2.8]([2.6] [2.7]([7.4])) = [2.2]              { <n>[N2] <o>2}
[8.8]   [2.8]([2.7]([7.3]) [2.6]) = [2.2]              { <n>[N3] <o>2}
[8.9]   [2.8]([2.7]([7.3]) [2.7]([7.4])) = [2.8]([7.3][7.4])
                                                       { <n>[N4] <o>2}
[9.1]   [4.1]([7.5]) = <alt,2>(
                          <seq,3>(<skip> [3.2] [4.1]([2.7]([7.5])))
                          <seq,3>(<skip> [3.3] [3.1]([7.5])))
                                                               { <o>3}
[9.2]   [4.2] = <hide>([5.1] <encaps>([5.2] <par,2>([4.1]([2.6])
                                       <sum>([7.9] [3.4]([7.9])))))
                                                               { <o>4}
```

5. CONCLUSION

We have defined a language meeting the criteria given for an intermediate language.

The language has the expressive power of PSF and provides the notion of free formats to include tool dependent information. This feature can be especially useful for creating so called *hooks* to the source language.

TIL is already being used as the core-language of the PSF toolkit. An interface to the ACP tools developed at PTT-research [110] is implemented, while interfaces to other tools are under construction.

We claim that other languages based on a combination of algebraic specification and process algebra can easily be translated to TIL. Of course we should extend the TIL language with the specific process operators used in other languages, however this can easily be done. Thus we reveal the possibilities of the PSF toolkit for languages as LOTOS [67], PSF/C [11] and μ-CRL [51].

APPENDIX A. ACTION RULES

In this appendix we will define the operational semantics for the process definition part of TIL with the aid of so-called *action rules*. Action rules in ACP are introduced in [47].

A.1. PROCESS DEFINITIONS

A process definition in general looks as follows:

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

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

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

A.2. ACTION RULES FOR TIL

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

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

$x \xrightarrow{a} \surd$ means that the process expression represented by x can terminate successfully by executing the atomic action $[a]$. The special symbol $\surd$ can be looked upon as a symbol indicating successful termination of a process. It is not a process expression.

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

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

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

- $\underline{x}$: : a list of process variables $(x_1 \ldots x_n)$.
  $\underline{x}[t/x_i]$: a substitution of term t at position i in list $\underline{x}$.
  $\underline{x}\backslash x_i$: the list obtained by deleting the element at position i from list $\underline{x}$.
- *Dom(d)* : the set associated with variable d, as defined in the variable declaration of d.

  $|Dom(d)|$: the number of elements in the domain of d.

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

- *<par>*
  - $a \mid b = c$ means that the communication between a and b has been defined to be c.
- *<encaps>*
  - H : the set of atomic actions that have to be encapsulated.
- *<hide>*
  - I : the set of atomic actions that have to be renamed into *skip*.
- *rec.*
  - $\underline{u} \in \underline{D}$ means $u_1 \in D_1, u_2 \in D_2, \ldots, u_n \in D_n$
    - $\underline{u} = (u_1, u_2, \ldots, u_n)$
    - $\underline{D} = (D_1, D_2, \ldots, D_n)$
    - D_i is a sort.
  - $y(\underline{u})$: a process expression with a list of terms $\underline{u} \in \underline{D}$ as parameters.
  - X : a process name declared in a process *declaration* tuple as $X\ n\ D_1\ D_2\ \ldots\ D_n$
  - $X(\underline{u}) = y(\underline{u})$: an equation from a *definition* tuple.
- *<sum>*
  - d is a variable.

atom	$a \xrightarrow{a} \surd$
<alt> 1	$\dfrac{x_i \xrightarrow{a} x' \quad (1 \le i \le n)}{\langle a,n\rangle(x_1 \ldots x_n) \xrightarrow{a} x'}$
<alt> 2	$\dfrac{x_i \xrightarrow{a} \surd \quad (1 \le i \le n)}{\langle a,n\rangle(x_1 \ldots x_n) \xrightarrow{a} \surd}$
<seq> 1	$\dfrac{x_1 \xrightarrow{a} x'}{\langle s,n\rangle(x_1 \ldots x_n) \xrightarrow{a} \langle s,n\rangle(x'\ x_2 \ldots x_n)}$
<seq> 2	$\dfrac{x_1 \xrightarrow{a} \surd \quad (n>2)}{\langle s,n\rangle(x_1 \ldots x_n) \xrightarrow{a} \langle s,n-1\rangle(x_2 \ldots x_n)}$

<seq> 3	$\dfrac{x_1 \xrightarrow{a} \surd}{\text{<s,2>}(x_1\ x_2) \xrightarrow{a} x_2}$
<par> 1	$\dfrac{x_i \xrightarrow{a} x' \quad \mathit{(1 \leq i \leq n)}}{\text{<p,n>}(x_1 \ldots x_n) \xrightarrow{a} \text{<p,n>}(\underline{x}[x'/x_i])}$
<par> 2	$\dfrac{x_i \xrightarrow{a} \surd \quad \mathit{(1 \leq i \leq n;\ n>2)}}{\text{<p,n>}(x_1 \ldots x_n) \xrightarrow{a} \text{<p,n>}(\underline{x} \backslash x_j)}$
<par> 3	$\dfrac{x_i \xrightarrow{a} \surd \quad \mathit{(1 \leq i,j \leq 2;\ i \neq j)}}{\text{<p,2>}(x_1\ x_2) \xrightarrow{a} x_j}$
<par> 4	$\dfrac{x_i \xrightarrow{a} x';\ x_j \xrightarrow{b} x'';\ a \mid b=c \quad \mathit{(1 \leq i \leq n;\ 1 \leq j \leq n;\ i \neq j)}}{\text{<p,n>}(x_1 \ldots x_n) \xrightarrow{c} \text{<p,n>}(\underline{x}[x'/x_i][x''/x_j])}$
<par> 5	$\dfrac{x_i \xrightarrow{a} x';\ x_j \xrightarrow{b} \surd;\ a \mid b=c \quad \mathit{(1 \leq i \leq n;\ 1 \leq j \leq n;\ n>2;\ i \neq j)}}{\text{<p,n>}(x_1 \ldots x_n) \xrightarrow{c} \text{<p,n-1>}(\underline{x}[x'/x_i] \backslash x_j)}$
<par> 6	$\dfrac{x_i \xrightarrow{a} x';\ x_j \xrightarrow{b} \surd;\ a \mid b=c \quad \mathit{(1 \leq i \leq n;\ 1 \leq j \leq n;\ i \neq j)}}{\text{<p,2>}(x_1\ x_2) \xrightarrow{c} x'}$
<par> 7	$\dfrac{x_i \xrightarrow{a} \surd;\ x_j \xrightarrow{b} \surd;\ a \mid b=c \quad \mathit{(1 \leq i \leq n;\ 1 \leq j \leq n;\ i \neq j;\ n>3)}}{\text{<p,n>}(x_1 \ldots x_n) \xrightarrow{c} \text{<p,n-2>}(\underline{x} \backslash x_i \backslash x_j)}$
<par> 8	$\dfrac{x_i \xrightarrow{a} \surd;\ x_j \xrightarrow{b} \surd;\ a \mid b=c \quad \mathit{(1 \leq i \leq 3;\ 1 \leq j \leq 3;\ i \neq j)}}{\text{<p,3>}(x_1\ x_2\ x_3) \xrightarrow{c} \underline{x} \backslash x_i \backslash x_j}$
<par> 9	$\dfrac{x_i \xrightarrow{a} \surd;\ x_j \xrightarrow{b} \surd;\ a \mid b=c \quad \mathit{(1 \leq i \leq 2;\ 1 \leq j \leq 2;\ i \neq j)}}{\text{<p,n>}(x_1 \ldots x_n) \xrightarrow{c} \surd}$
<encaps> 1	$\dfrac{x \xrightarrow{a} x';\ a \notin H}{\text{<encaps>}(H\ x) \xrightarrow{a} \text{<encaps>}(H\ x')}$
<encaps> 2	$\dfrac{x \xrightarrow{a} \surd;\ a \notin H}{\text{<encaps>}(H\ x) \xrightarrow{a} \surd}$
<hide> 1	$\dfrac{x \xrightarrow{a} x';\ a \in I}{\text{<hide>}(I\ x) \xrightarrow{\text{<skip>}} \text{<hide>}(I\ x')}$

<hide> 2	$\dfrac{x \xrightarrow{a} \surd;\ a \in I}{\langle\text{hide}\rangle(I\ x) \xrightarrow{\langle\text{skip}\rangle} \surd}$
<hide> 3	$\dfrac{x \xrightarrow{a} x';\ a \notin I}{\langle\text{hide}\rangle(I\ x) \xrightarrow{a} \langle\text{hide}\rangle(I\ x')}$
<hide> 4	$\dfrac{x \xrightarrow{a} \surd;\ a \notin I}{\langle\text{hide}\rangle(I\ x) \xrightarrow{a} \surd}$
rec. 1	$\dfrac{y(\underline{u}) \xrightarrow{a} y' \quad (\underline{u} \in \underline{D};\ X(\underline{u}) = y(\underline{u}))}{X(\underline{u}) \xrightarrow{a} y'}$
rec. 2	$\dfrac{y(\underline{u}) \xrightarrow{a} \surd \quad (\underline{u} \in \underline{D};\ X(\underline{u}) = y(\underline{u}))}{X(\underline{u}) \xrightarrow{a} \surd}$
<sum> 1	$\dfrac{x(u) \xrightarrow{a} x' \quad (u \in Dom(d))}{\langle\text{sum}\rangle(d\ \ x(d)) \xrightarrow{a} x'}$
<sum> 2	$\dfrac{x(u) \xrightarrow{a} \surd \quad (u \in Dom(d))}{\langle\text{sum}\rangle(d\ \ x(d)) \xrightarrow{a} \surd}$
<merge> 1	$\dfrac{x(u) \xrightarrow{a} x' \quad (\lvert Dom(d)\rvert>1;\ u \in Dom(d);\ Dom(d')=Dom(d)\backslash\{u\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{a} \langle p,2\rangle(x'\ \langle\text{merge}\rangle(d'\ x(d')))}$
<merge> 2	$\dfrac{x(u) \xrightarrow{a} \surd \quad (\lvert Dom(d)\rvert>1;\ u \in Dom(d);\ Dom(d')=Dom(d)\backslash\{u\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{a} \langle\text{merge}\rangle(d'\ x(d'))}$
<merge> 3	$\dfrac{x(u) \xrightarrow{a} x' \quad (Dom(d)=\{u\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{a} x'}$
<merge> 4	$\dfrac{x(u) \xrightarrow{a} \surd \quad (Dom(d)=\{u\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{a} \surd}$
<merge> 5	$\dfrac{x(u) \xrightarrow{a} x';\ x(v) \xrightarrow{b} x'';\ a \mid b=c \quad (\lvert Dom(d)\rvert>2;\ u,v \in Dom(d);\ Dom(d')=Dom(d)\backslash\{u,v\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{c} \langle p,3\rangle(x'\ x''\ \langle\text{merge}\rangle(d'\ x(d')))}$
<merge> 6	$\dfrac{x(u) \xrightarrow{a} x';\ x(v) \xrightarrow{b} \surd;\ a \mid b=c \quad (\lvert Dom(d)\rvert>2;\ u,v \in Dom(d);\ Dom(d')=Dom(d)\backslash\{u,v\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{c} \langle p,2\rangle(x'\ \langle\text{merge}\rangle(d'\ x(d')))}$
<merge> 7	$\dfrac{x(u) \xrightarrow{a} \surd;\ x(v) \xrightarrow{b} \surd;\ a \mid b=c \quad (\lvert Dom(d)\rvert>2;\ u,v \in Dom(d);\ Dom(d')=Dom(d)\backslash\{u,v\})}{\langle\text{merge}\rangle(d\ x(d)) \xrightarrow{c} \langle\text{merge}\rangle(d'\ x(d'))}$

<merge> 8	$\dfrac{x(u) \xrightarrow{a} x';\ x(v) \xrightarrow{b} x'';\ a\vert b=c \quad (Dom(d)=\{u,v\})}{\text{<merge>}(d\ x(d)) \xrightarrow{c} \text{<p,2>}(x'\ x'')}$
<merge> 9	$\dfrac{x(u) \xrightarrow{a} x';\ x(v) \xrightarrow{b} \surd;\ a\vert b=c \quad (Dom(d)=\{u,v\})}{\text{<merge>}(d\ x(d)) \xrightarrow{c} x'}$
<merge> 10	$\dfrac{x(u) \xrightarrow{b} \surd;\ x(v) \xrightarrow{a} \surd;\ a\vert b=c \quad (Dom(d)=\{u,v\})}{\text{<merge>}(d\ x(d)) \xrightarrow{c} \surd}$
<if> 1	$\dfrac{x \xrightarrow{a} x' \quad (s=t)}{\text{<if>}(s=t\ \ x) \xrightarrow{a} x'}$
<if> 2	$\dfrac{x \xrightarrow{a} \surd \quad (s=t)}{\text{<if>}(s=t\ \ x) \xrightarrow{a} \surd}$
<case> 1	$\dfrac{x_i \xrightarrow{a} x';\ s=t_i \quad (1 \le i \le n)}{\text{<case,n>}(s\ t_1\ x_1 \dots t_n\ x_n) \xrightarrow{a} x'}$
<case> 2	$\dfrac{x_i \xrightarrow{a} \surd;\ s=t_i \quad (1 \le i \le n)}{\text{<case,n>}(s\ t_1\ x_1 \dots t_n\ x_n) \xrightarrow{a} \surd}$

figure A.1 Table of action relations.

A.3. Process Semantics

Now that we have defined the action relations for TIL we are able to assign a semantics to processes. In this case we define *bisimulation* [94] on top of these action relations.

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

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

If there exists a bisimulation R on process expressions with pRq, then p and q are called *bisimilar*, notation $p \underline{\leftrightarrow} q$. Bisimulation is an equivalence relation.

Process terms are interpreted in the semantic domain that is obtained by taking process expressions modulo bisimulation.

Chapter 5

SPECIFICATION AND VERIFICATION OF CIM-ARCHITECTURES

Flexibility of a manufacturing system implies that it must be possible to reorganise the configuration of the system's components efficiently and correctly. To avoid costly redesign, we have the need for a formal description technique for specifying the (co)operation of the components. Process algebra will be shown to be expressive enough to specify, and even verify, the correct functioning of such a system. This will be demonstrated by formally specifying and verifying two workcells, which can be viewed as units of a small number of cooperating machines. The specifications will be provided in PSF, while the verifications will take place in the framework of ACP.

1. INTRODUCTION

One can speak of *Computer Integrated Manufacturing* (CIM) if the computer is used in all phases of the production of some industrial product. In this chapter we will focus on the design of the product-flow and the information-flow, which occurs when products are actually produced. Topics like product-development, marketing and management are beyond the scope of this chapter. The technique used in this chapter is based on a theory for concurrency, called *process algebra* (see [14]). It can be used to describe the total phase of manufacturing, from the ordering of raw materials up to the shipping of the products which are made from these materials. During this process many machines are used, which can operate independently, but often depend on the correct operation of each other. Providing a correct functioning of the total of all machines, computers and transport-services is not a trivial exercise. Before actually building such a system (a *CIM-architecture*) there must be some design. Such a specification, when validated, describes a properly functioning system. The current trend towards *Flexible Manufacturing Systems* (FMS) introduces the need for a tool, able to validate a new design of a plant, before implementing it. The possibilities to use methods developed in process algebra for *specification* and *verification* of concurrent systems are described in this chapter.

From a high level of view, a plant can be seen as constructed from several concurrently operating *workcells* (W1-W5 in figure 1). Every workcell is responsible for some well-defined part of the manufacturing process, e.g. filling and capping a number of milk bottles. The various workcells are connected to each other via some transport-service, which manages input and output of goods for the workcells (the *logistics*).

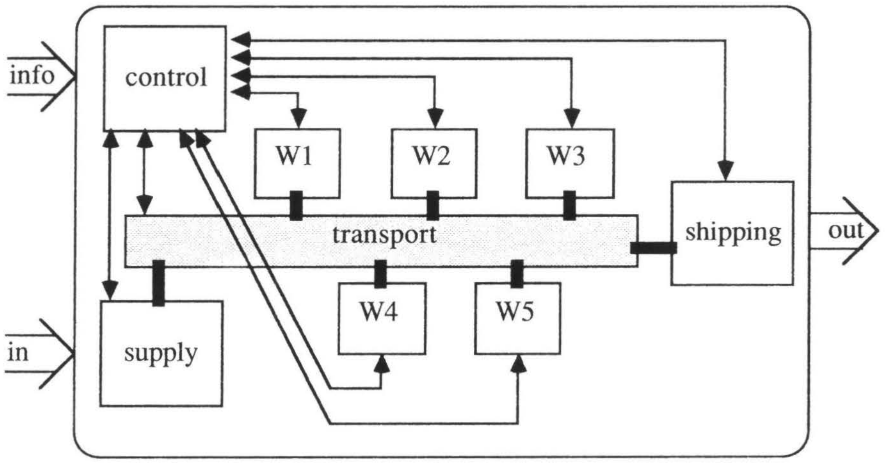


Figure 1 A sample architecture of a plant

Of course some supervisor (*control*) must keep track of the (co)operation of all workcells. This control has connections to all other components of the plant, along which commands and status-reports are transmitted. The components labeled *supply* and *shipping* are used to store raw materials and processed goods. Seen from a lower level, each workcell is constructed from a number of basic components which can perform one function, e.g. drilling a hole or assembling two parts. For controlling the communication with the outside and to instruct the various components of the workcell, each workcell has a *workcell-controller*. Also some simple transport-system must be present to transport the products within the workcell (see figure 2).

The description of the components of some workcell can be given using PSF. When abstracting from the internal actions of that workcell, it is possible to determine its external behaviour. At the high level view on the flow of products, we are only interested in the products which enter the workcell and the products leaving it. Also at the high level view on the flow of information, we only look at the commands we give a workcell to produce or process a number of products and the status-reports sent back.

The simple two level view on a manufacturing process expressed above, can be refined into a multi layered model, as is done in e.g. [29].

As an illustration of the technique we specify two workcells in the specification formalism PSF and verify their correctness with the theory ACP_τ (see [23]). The first workcell is a very simple one, able to produce and process one kind of product. The second one is more involved. It has the possibility to process some input product either correctly or faultily. Part of the workcell is a quality-check tool, which decides upon rejecting the product or not.

One should notice that in process algebra as we use it here, no real-time aspects are captured. So the important notions of *efficiency* (maximal productivity of the machines) and *tuning* (synchronization of the speed of the machines) cannot be modeled. For real time extensions of process algebra, see [6].

This chapter is partially based on discussions with F. Biemans, and inspired by his article [30], who used the specification language LOTOS (see [67]) to describe CIM-architectures. Other applications of theories for concurrency to CIM can be found in [72] and [73].

2. A Simple Workcell

In this section a simple workcell will be specified and verified, which consists of four components. This workcell is identical to the one described in [30].

2.1. Specification

2.1.1. Basic Datatypes

The following modules are needed for the specification of both workcells. They define *booleans, naturals, bounded naturals* and *queues*. In the module *bounded-naturals* a set of naturals is defined, containing all naturals up to some upper bound N. This upper bound is a parameter of the specification. It determines the maximum number of products the workcell can produce in one drive. The module *queues* also has a parameter, which determines the type of items to be queued.

```
data module booleans
begin

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

end booleans


data module naturals
begin

  exports
    begin
      sorts nat
      functions
        0   :             -> nat
        s   : nat         -> nat
        add : nat # nat -> nat
  end

  imports
    booleans

  variables
    n, m : -> nat
  equations
  [1]  add(0, n) = n
  [2]  add(s(n), m) = s(add(n, m))

end naturals


process module bounded-naturals
begin

  parameters
    max
      begin
        functions
          N : -> nat
      end max

  exports
    begin
      sets of nat
        bounded-nat = nat \ {add(n, s(N)) | n in nat}
    end

  imports
    naturals

end bounded-naturals
```

```
data module queues
begin

  parameters
    items
      begin
        sorts item
    end items

  exports
    begin
      sorts queue
      functions
        empty-queue :                 -> queue
        add         : item # queue -> queue
        add-back    : item # queue -> queue
    end

  variables
    i, j : -> item
    q    : -> queue
  equations
  [1] add-back(i, empty-queue) = add(i, empty-queue)
  [2] add-back(i, add(j, q)) = add(j, add-back(i, q))

end queues
```

2.1.2. General Description

The Workcell consists of four components (see figure 2).

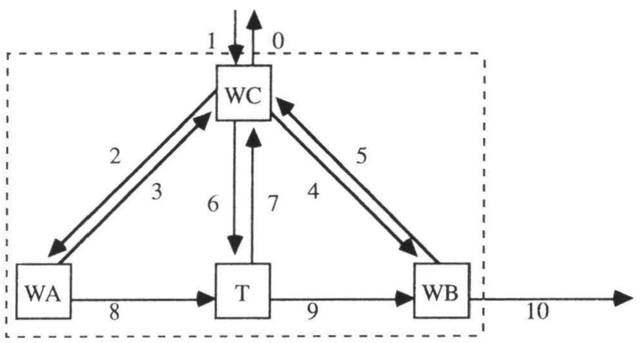


Figure 2 A simple workcell

Workstation A (*WA*) produces a product (*p1*) and offers this to the Transport service (*T*). Then the product is transported to Workstation B (*WB*), which processes the product and outputs it to the environment. The Workcell Controller (*WC*) receives a command from the environment to produce a number of products, then controls the operating of the other components and reports a ready-status back to the environment. So the total of the four components can be viewed as one workcell, producing and processing a number of products. The aim is to specify the components in such a way that the workcell behaves as desired.

2.1.3. Datatypes Specific to the Workcell

The four components are connected by 11 ports. Some ports are used to transmit data (the ports 0 through 7), while others are used to exchange products (the ports 8 through 10). Three ports are connected to the environment (the ports 0, 1 and 10).

```
data module ports
begin

  exports
    begin
      sorts data-ports, product-ports
      functions
        port0  : -> data-ports
        port1  : -> data-ports
        port2  : -> data-ports
        port3  : -> data-ports
        port4  : -> data-ports
        port5  : -> data-ports
        port6  : -> data-ports
        port7  : -> data-ports
        port8  : -> product-ports
        port9  : -> product-ports
        port10 : -> product-ports
    end

end ports
```

The sort *products* contains all products that are produced and processed within the workstation (or the complete factory). It is a parameter of the system and contains at least the products product1 (*p1*) and the processed product1 (*proc(p1)*).

```
data module products
begin

  parameters
    products
      begin
        sorts products
        functions
          p1   :            -> products
          proc : products -> products
      end products

end products
```

Several kinds of data have to be transmitted throughout the workcell. Via the ports 1, 2 and 4 a non-negative integer (*produce(n)*) can be sent to indicate that the receiver has to *produce* (or process) *n* products. A ready message (*ready*) is sent back over the ports 0, 3 and 5 to indicate that the corresponding component has fulfilled its task. Over port 6 the Workcell Controller can send a transport command (*transport*) to the Transport Service, indicating that one product has to be transported from *WA* to *WB*. If this is done, an arrival-message (*arrival*) is sent back via port 7. These messages are all collected in the sort *messages*.

```
data module messages
begin

  exports
    begin
      sorts messages
      functions
        produce   : nat          -> messages
        ready     :              -> messages
        transport :              -> messages
        arrival   :              -> messages
    end

  imports
    naturals

end messages
```

The product-ports are used to transport products, whereas the data-ports are used to transmit messages. For both kinds of exchange, we use read-send communication. Furthermore we construct sets for encapsulation and abstraction, containing all actions along internal ports.

```
process module communication
begin

  exports
    begin
      atoms
        r : product-ports # products
        s : product-ports # products
        c : product-ports # products
        r : data-ports # messages
        s : data-ports # messages
        c : data-ports # messages
      sets of data-ports
        internal-data-ports =
              {port2, port3, port4, port5, port6, port7}
      of product-ports
        internal-product-ports = {port8, port9}
      of atoms
        I = { c(dp,m), c(pp,p) |
              dp in internal-data-ports, m in messages,
              pp in internal-product-ports, p in products }
        H = { r(dp,m), s(dp,m), r(pp,p), s(pp,p) |
                  dp in internal-data-ports, m in messages,
                  pp in internal-product-ports, p in products }
      end

    imports
      ports, products, messages

    communications
      r(dp,m) | s(dp,m) = c(dp,m)
        for dp in data-ports, m in messages
      r(pp,p) | s(pp,p) = c(pp,p)
        for pp in product-ports, p in products

  end communication
```

2.1.4. Workstation A

The first component to be described is Workstation A. It receives via port 2 the command to produce *n* times product *p1*. The range of the summation is the set *bounded-nat,* which contains the naturals up to *N*. This bound determines the maximum number of products the workcell can deal with in one drive. Then it executes this command by producing *n* products (XA(n)) and sends a ready-status message at port 3. Then WA starts all over. If WA was commanded to produce zero products, XA(0) just ends after doing the internal action *skip*. If a positive number of products has to be produced (XA(s(n))), this is done by producing one product, followed by the production of *n* products.

```
process module Workstation-A
begin

  exports
    begin
      processes
        WA
    end

  imports
    communication, bounded-naturals

  processes
    XA : nat

  variables
    n : -> nat
  definitions
    WA        = sum(n in bounded-nat,
                  r(port2, produce(n)) . XA(n) . s(port3, ready) . WA)
    XA(0)     = skip
    XA(s(n))  = s(port8, p1) . XA(n)

end Workstation-A
```

2.1.5. Workstation B

Workstation B has almost the same definition as Workstation A. It accepts the command to process *n* products via port 4, processes *n* products (XB(n)), sends a ready-status message and starts all over. The processing of *n* products is achieved by repeatedly receiving an arbitrary product *p* at port 9 and sending the processed version of this product to port 10.

```
process module Workstation-B
begin

  exports
    begin
      processes
        WB
    end

  imports
    communication, bounded-naturals

  processes
    XB : nat

  variables
    n : -> nat
  definitions
    WB         = sum(n in bounded-nat,
                 r(port4, produce(n)) . XB(n) . s(port5, ready) . WB)
    XB(0)      = skip
    XB(s(n))   = sum(p in products,
                     r(port9, p) . s(port10, proc(p)) . XB(n))

end Workstation-B
```

2.1.6. Transport Service

The Transport service (T) can be seen as a FIFO-queue. It is indexed with its contents, a queue of products. The transport system either has an empty queue, or contains elements. If the queue is empty, T can receive a transport-command via port 6 and then it receives some product via port 8. Next the transport service behaves as the transport service with one element in its queue. It is also possible to receive the product first and then receive the transport-command. If the queue was not empty, the Transport service has both options as mentioned for the empty queue, but it also has the option to send an element out of the queue at port 9. Then the arrival of this element is reported to the Workcell Controller and the element is deleted from the queue.

```
process module Transport-Service
begin

  exports
    begin
      processes
        T : product-queue
    end

  imports
    communication,
    queues
      {items bound by
         [item -> products]
       to products
       renamed by
         [queue -> product-queue]}

  variables
    q : -> product-queue
    r : -> products
  definitions
    T(empty-queue) = r(port6, transport) .
                       sum(p in products,
                         r(port8, p) . T(add-back(p, empty-queue)))+
                     sum(p in products,
                       r(port8, p) . r(port6, transport) .
                       T(add-back(p, empty-queue)))
    T(add(r, q))   = r(port6, transport) .
                       sum(p in products,
                         r(port8, p) . T(add-back(p, add(r, q)))) +
                     sum(p in products,
                       r(port8, p) . r(port6, transport) .
                       T(add-back(p, add(r, q)))) +
                    s(port9, r) . s(port7, arrival) . T(q)

end Transport-Service
```

2.1.7. Workcell Controller

The Workcell Controller (WC) controls the communication with the environment and the interaction of the other components. It receives via port 1 the command to produce and process *n* products. Then it commands Workstation B via port 4 to process *n* products and goes into state D(n) were *n* times product1 is produced and transported. Then finally it receives a ready-status message from WB via port 5 and sends *ready* to the environment, returning to its initial state. The production and transport of *n* products is done in D(n). It repeatedly commands via port 2 Workstation A to produce one single product. If this is done a ready message is received at port 3 and a transport command is sent at port 6. If the product has arrived at Workstation B, an arrival message is received at port 7.

```
process module Workcell-Controller
begin

  exports
    begin
      processes
        WC
    end

  imports
    communication, bounded-naturals

  processes
    D : nat

  variables
    n : -> nat
  definitions
    WC      = sum(n in bounded-nat,
                  r(port1, produce(n)) . s(port4, produce(n)) . D(n).
                  r(port5, ready) . s(port0, ready) . WC)
    D(0)    = skip
    D(s(n)) = s(port2, produce(s(0))) . r(port3, ready) .
                 s(port6, transport) . r(port7, arrival) . D(n)

end Workcell-Controller
```

2.1.8. The Workcell

The concurrent operation of these four components can be considered as the specification of the whole workcell. Notice that the Transport service has to start with an empty queue.

```
process module Workcell
begin

  exports
    begin
      processes
        W
    end

  imports
    Workstation-A, Workstation-B,
    Transport-Service, Workcell-Controller

  definitions
    W = hide(I, encaps(H, WC || T(empty-queue) || WA || WB))

end Workcell
```

2.2. CORRECTNESS

2.2.1. Preliminaries

In order to prove correctness of the protocol we will need some extra proof rules from [14]. The first one is RDP, the Recursive Definition Principle, which states that every recursive specification has a solution. The rule RSP, the Recursive Specification Principle, states that every guarded specification has at most one solution. Together they provide uniqueness of the solution of a guarded recursive specification.

The Expansion Theorem is used to expand a merge into its subterms. For $n \geq 3$ it reads:

$$\boxed{x_1 \parallel \ldots \parallel x_n = \sum_{1 \leq i \leq n} x_i \mathbin{\|\!\!\lfloor} \Big(\mathop{\parallel}_{1 \leq j \leq n\ j \neq i} x_j \Big) + \sum_{1 \leq i < j \leq n} (x_i \mid x_j) \mathbin{\|\!\!\lfloor} \Big(\mathop{\parallel}_{1 \leq k \leq n\ k \neq i,j} x_k \Big)}$$

2.2.2. Intended Behaviour

When designing the workcell, we had in mind some idea about its external behaviour. It receives a command at port 1, which indicates the number of products that has to be produced, then these products are produced and offered at port 10 and finally a ready message is offered at port 0 and we return to the starting state. This intended behaviour is specified in the following module.

```
process module Workcell-Behaviour
begin

  exports
    begin
      processes
        V
    end

  imports
    communication, bounded-naturals

  processes
    E : nat

  variables
    n : -> nat
  definitions
    V        = sum(n in bounded-nat, r(port1, produce(n)) . E(n) . V)
    E(0)     = s(port0, ready)
    E(s(n))  = s(port10, proc(p1)) . E(n)

end Workcell-Behaviour
```

Now, using RDP, let v and w be solutions of the specifications of V and W. A proof that the processes v and w are equal can be seen as a verification that the specification of W is correct with respect to its intended external behaviour.

2.2.3. The Workcell is Correct

We prove the following theorem.

THEOREM 1 *The specification of the workcell is correct.*

$$ACP_{\tau} + RDP + RSP + ET \;\vdash\; v=w$$

PROOF

The proof consists of a series of successive expansions. All atoms that do not communicate yield deadlock, because they are encapsulated. The atoms that do communicate are underlined. All actions that are not abstracted from are boldfaced.

Some shorthands are used to obtain a more concise verification. For example the expression $\Sigma r1(n)\ldots$ is an abbreviation of *sum(n in bounded-nat, r(port1, produce(n)...).* Abstraction is denoted by τ_I and encapsulation by ∂_H. We interpret the internal action *skip* by the silent step τ from process algebra. The empty queue is denoted by λ, and adding to a queue is done with the $*$ operator.

$$\begin{aligned}
W &= \tau_I\, \partial_H\,(WC \,||\, T^{\lambda} \,||\, WA \,||\, WB) \\
&= \tau_I\, \partial_H\,((\underline{\Sigma r1(n)} \,.\, s4(n) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,|| \\
&\qquad (r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,|| \\
&\qquad (\Sigma r2(n) \,.\, XA^n \,.\, s3(r) \,.\, WA) \,|| \\
&\qquad (\Sigma r4(n) \,.\, XB^n \,.\, s5(r) \,.\, WB)) \\
&= \mathbf{\Sigma r1(n)} \,.\, \tau_I\, \partial_H\,((\underline{s4(n)} \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,|| \\
&\qquad (r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,|| \\
&\qquad (\Sigma r2(n) \,.\, XA^n \,.\, s3(r) \,.\, WA) \,|| \\
&\qquad (\underline{\Sigma r4(n)} \,.\, XB^n \,.\, s5(r) \,.\, WB)) \\
&= \mathbf{\Sigma r1(n)} \,.\, \tau_I\,\,(c4(n) \,.\, \partial_H\,((D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,|| \\
&\qquad (r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,|| \\
&\qquad (\Sigma r2(n) \,.\, XA^n \,.\, s3(r) \,.\, WA) \,|| \\
&\qquad (XB^n \,.\, s5(r) \,.\, WB)))
\end{aligned}$$

Now let, for $n \in N$

$$\begin{aligned}
K^n &= \tau_I \partial_H\,((D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,|| \\
&\qquad (r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,|| \\
&\qquad (\Sigma r2(n) \,.\, XA^n \,.\, s3(r) \,.\, WA) \,|| \\
&\qquad (XB^n \,.\, s5(r) \,.\, WB)), \text{ then} \\
K^0 &= \tau_I \partial_H\,((\tau \,.\, \underline{r5(r)} \,.\, s0(r) \,.\, WC) \,|| \\
&\qquad (r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,|| \\
&\qquad (\Sigma r2(n) \,.\, XA^n \,.\, s3(r) \,.\, WA) \,|| \\
&\qquad (\tau \,.\, \underline{s5(r)} \,.\, WB))
\end{aligned}$$

$= \tau \,.\, \tau_I \partial_H ((\underline{s0(r)} \,.\, WC) \,||$

$(r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,||$

$(\Sigma r2(n) \,.\, XA^n \,.\, s3(r) \,.\, WA) \,||$

$(WB))$

$= \tau \,.\, \mathbf{s0(r)} \,.\, W$

$K^{s(n)} = \tau_I \partial_H ((\underline{s2(1)} \,.\, r3(r) \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$

$(r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(r8(p) \,.\, r6(t) \,.\, T^p)) \,||$

$(\underline{\Sigma r2(n)} \,.\, XA^{s(n)} \,.\, s3(r) \,.\, WA) \,||$

$(\Sigma r9(p) \,.\, s10(proc(p)) \,.\, XB^n \,.\, s5(r) \,.\, WB))$

$= \tau_I \,(c2(1) \,.\, \partial_H ((r3(r) \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$

$(r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(\underline{r8(p)} \,.\, r6(t) \,.\, T^p)) \,||$

$(\underline{s8(p1)} \,.\, XA^0 \,.\, s3(r) \,.\, WA) \,||$

$(\Sigma r9(p) \,.\, s10(proc(p)) \,.\, XB^n \,.\, s5(r) \,.\, WB)))$

$= \tau \,.\, \tau_I \,(c8(p1) \,.\, \partial_H ((\underline{r3(r)} \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$

$(r6(t) \,.\, T^{p1}) \,||$

$(\tau \,.\, \underline{s3(r)} \,.\, WA) \,||$

$(\Sigma r9(p) \,.\, s10(proc(p)) \,.\, XB^n \,.\, s5(r) \,.\, WB)))$

$= \tau \,.\, \tau_I \,(c3(r) \,.\, \partial_H ((\underline{s6(t)} \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$

$(\underline{r6(t)} \,.\, T^{p1}) \,||$

$(WA) \,||$

$(\Sigma r9(p) \,.\, s10(proc(p)) \,.\, XB^n \,.\, s5(r) \,.\, WB)))$

$= \tau \,.\, \tau_I \,(c6(t) \,.\, \partial_H ((r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$ (**)

$(r6(t) \,.\, \Sigma(r8(p) \,.T^{p^*p1}\ ) + \Sigma(r8(p).r6(t) \,.T^{p^*p1}\ ) + \underline{s9(p1)} \,.\, s7(ar) \,.\, T^\lambda) \,||$

$(WA) \,||$

$(\underline{\Sigma r9(p)} \,.\, s10(proc(p)) \,.\, XB^n \,.\, s5(r) \,.\, WB)))$

$= \tau \,.\, \tau_I \,(c9(p1) \,.\, \partial_H ((\underline{r7(ar)} \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$

$(\underline{s7(ar)} \,.\, T^\lambda) \,||$

$(WA) \,||$

$(\underline{s10(proc(p1))} \,.\, XB^n \,.\, s5(r) \,.\, WB)))$

$= \tau \,.\, (\tau_I \,(c7(ar) \,.\, \partial_H ((D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$

$(T^\lambda) \,||$

$(WA) \,||$

$(s10(proc(p1)) \,.\, XB^n \,.\, s5(r) \,.\, WB))) +$

$\mathbf{s10(proc(p1))} \,.\, K^n)$

Now let, for $n \in N$

$$\begin{aligned}
L^n &= \tau_I\, \partial_H ((D^n . r5(r) . s0(r) . WC) \parallel \\
&\quad (T^\lambda) \parallel \\
&\quad (WA) \parallel \\
&\quad (s10(proc(p1)) . XB^n . s5(r) . WB)), \text{ then} \\
L^0 &= \tau_I\, \partial_H ((\tau . r5(r) . s0(r) . WC) \parallel \\
&\quad (T^\lambda) \parallel \\
&\quad (WA) \parallel \\
&\quad (\underline{s10(proc(p1))} . \tau . s5(r) . WB)) \\
&= \mathbf{s10(proc(p1))} . \tau_I\, \partial_H ((\tau . \underline{r5(r)} . s0(r) . WC) \parallel \\
&\quad (T^\lambda) \parallel \\
&\quad (WA) \parallel \\
&\quad (\tau . \underline{s5(r)} . WB)) + \\
&\quad \tau . \mathbf{s10(proc(p1))} . \tau_I\, \partial_H ((\underline{r5(r)} . s0(r) . WC) \parallel \\
&\quad (T^\lambda) \parallel \\
&\quad (WA) \parallel \\
&\quad (\tau . \underline{s5(r)} . WB)) \\
&= \tau . \mathbf{s10(proc(p1))} . \tau_I\, (c5(r) . \partial_H ((\underline{s0(r)} . WC) \parallel \qquad \text{[using T2]} \\
&\quad (T^\lambda) \parallel \\
&\quad (WA) \parallel \\
&\quad (WB))) \\
&= \tau . \mathbf{s10(proc(p1)) . s0(r)} . W \\
L^{s(n)} &= \tau_I\, \partial_H ((\underline{s2(1)} . r3(r) . s6(t) . r7(ar) . D^n . r5(r) . s0(r) . WC) \parallel \\
&\quad (r6(t) . \Sigma(r8(p) . T^p) + \Sigma(r8(p) . r6(t) . T^p)) \parallel \\
&\quad (\Sigma \underline{r2(n)} . XA^n . s3(r) . WA) \parallel \\
&\quad (\underline{s10(proc(p1))} . XB^{s(n)} . s5(r) . WB)) \\
&= \tau_I\, (c2(1) . \partial_H ((r3(r) . s6(t) . r7(ar) . D^n . r5(r) . s0(r) . WC) \parallel \\
&\quad (r6(t) . \Sigma(r8(p) . T^p) + \Sigma(\underline{r8(p)} . r6(t) . T^p)) \parallel \\
&\quad (\underline{s8(p1)} . XA^0 . s3(r) . WA) \parallel \\
&\quad (\underline{s10(proc(p1))} . XB^{s(n)} . s5(r) . WB))) + \\
&\quad \mathbf{s10(proc(p1))} . \tau_I\, \partial_H ((\underline{s2(1)} . r3(r) . s6(t) . r7(ar) . D^n . r5(r) . s0(r) . WC) \parallel \\
&\quad (r6(t) . \Sigma(r8(p) . T^p) + \Sigma(r8(p) . r6(t) . T^p)) \parallel \\
&\quad (\Sigma \underline{r2(n)} . XA^n . s3(r) . WA) \parallel \\
&\quad (XB^{s(n)} . s5(r) . WB))
\end{aligned}$$

$= \tau \,.\, \tau_I\,(c8(p1) \,.\, \partial_H\,((r3(r) \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(r6(t) \,.\, T^{p1}) \,||$
$(XA^0 \,.\, s3(r) \,.\, WA\,) \,||$
$(\underline{s10(proc(p1))} \,.\, XB^{s(n)} \,.\, s5(r) \,.\, WB))) \,+$
$\tau \,.\, \mathbf{s10(proc(p1))} \,.\, \tau_I\partial_H\,((r3(r) \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(\underline{r8(p)} \,.\, r6(t) \,.\, T^p)) \,||$
$(\underline{s8(p1)} \,.\, XA^0 \,.\, s3(r) \,.\, WA\,) \,||$
$(XB^{s(n)} \,.\, s5(r) \,.\, WB)) \,+$
$\mathbf{s10(proc(p1))} \,.\, \tau_I\,(c2(1) \,.\, \partial_H\,((r3(r) \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(\underline{r8(p)} \,.\, r6(t) \,.\, T^p)) \,||$
$(\underline{s8(p1)} \,.\, XA^0 \,.\, s3(r) \,.\, WA\,) \,||$
$(XB^{s(n)} \,.\, s5(r) \,.\, WB))$

(The first two summands in this expression come from the first summand in the previous expression. Axiom T2 states that the summation of the second and third summand equals the second summand.)

$= \tau \,.\, \tau_I\,(c8(p1) \,.\, \partial_H\,((\underline{r3(r)} \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(r6(t) \,.\, T^{p1}) \,||$
$(\tau \,.\, \underline{s3(r)} \,.\, WA\,) \,||$
$(\underline{s10(proc(p1))} \,.\, XB^{s(n)} \,.\, s5(r) \,.\, WB))) \,+$
$\tau \,.\, \mathbf{s10(proc(p1))} \,.\, \tau_I\partial_H\,((r3(r) \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(r6(t) \,.\, \Sigma(r8(p) \,.\, T^p) + \Sigma(\underline{r8(p)} \,.\, r6(t) \,.\, T^p)) \,||$
$(\underline{s8(p1)} \,.\, XA^0 \,.\, s3(r) \,.\, WA\,) \,||$
$(XB^{s(n)} \,.\, s5(r) \,.\, WB))$

$= \tau \,.\, \tau_I\,(c3(r) \,.\, \partial_H\,((\underline{s6(t)} \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(\underline{r6(t)} \,.\, T^{p1}) \,||$
$(WA\,) \,||$
$(\underline{s10(proc(p1))} \,.\, XB^{s(n)} \,.\, s5(r) \,.\, WB))) \,+$
$\tau \,.\, \mathbf{s10(proc(p1))} \,.\, \tau_I\,(c8(p1) \,.\partial_H\,((\underline{r3(r)} \,.\, s6(t) \,.\, r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(r6(t) \,.\, T^{p1}) \,||$
$(\tau \,.\, \underline{s3(r)} \,.\, WA\,) \,||$
$(XB^{s(n)} \,.\, s5(r) \,.\, WB)))$

$= \tau \,.\, \tau_I\,(c6(t) \,.\, \partial_H\,((r7(ar) \,.\, D^n \,.\, r5(r) \,.\, s0(r) \,.\, WC) \,||$
$(T^{p1}) \,||$
$(WA\,) \,||$
$(\underline{s10(proc(p1))} \,.\, XB^{s(n)} \,.\, s5(r) \,.\, WB))) \,+$

$$\begin{aligned}
&\tau\,.\ \mathbf{s10(proc(p1))}\ .\ \tau_I\,(c3(r)\,.\partial_H\,(\underline{(s6(t)}\ .\ r7(ar)\ .\ D^n\ .\ r5(r)\ .\ s0(r)\ .\ WC)\,\|\\
&\quad \underline{(r6(t)}\ .\ T^{p1})\,\|\\
&\quad (WA\ )\,\|\\
&\quad (XB^{s(n)}\ .\ s5(r)\ .\ WB)))\\
&= \tau\,.\ \mathbf{s10(proc(p1))}\ .\ \tau_I\ \partial_H\,((r7(ar)\ .\ D^n\ .\ r5(r)\ .\ s0(r)\ .\ WC)\,\|\\
&\quad (T^{p1})\,\|\\
&\quad (WA\ )\,\|\\
&\quad (\Sigma r9(p)\ .\ s10(proc(p))\ .\ XB^n\ .\ s5(r)\ .\ WB))\\
&= \tau\,.\ \mathbf{s10(proc(p1))}\ .\ K^{s(n)} \qquad \text{[see (**)]}
\end{aligned}$$

So the process *w* is a solution of the following system:

$$\boxed{\begin{array}{ll}
W & = \Sigma r1(n)\ .K^n\\
K^0 & = \tau\ .\ s0(r)\ .\ W\\
K^{s(n)} & = \tau\ .\ (\tau\ .\ L^n + s10(proc(p1))\ .\ K^n)\\
L^0 & = \tau\ .\ s10(proc(p1))\ .\ s0(r)\ .\ W\\
L^{s(n)} & = \tau\ .\ s10(proc(p1))\ .\ K^{s(n)}
\end{array}}$$

Specification 1

Now observe that we can replace the two equations for L by the following definition.

$$\boxed{\begin{array}{ll}
L^n & = \tau\ .\ s10(proc(p1))\ .\ K^n
\end{array}}$$

Substitution of L^n gives the following system.

$$\boxed{\begin{array}{ll}
W & = \Sigma r1(n)\ .K^n\\
K^0 & = \tau\ .\ s0(r)\ .\ W\\
K^{s(n)} & = \tau\ .\ (\tau\ .\ \tau\ .\ s10(proc(p1))\ .\ K^n\ +\\
 & \quad s10(proc(p1))\ .\ K^n)
\end{array}}$$

After applying T2 we get.

$$\boxed{\begin{array}{ll}
W & = \Sigma r1(n)\ .K^n\\
K^0 & = \tau\ .\ s0(r)\ .\ W\\
K^{s(n)} & = \tau\ .\ s10(proc(p1))\ .\ K^n)
\end{array}}$$

Specification 2

Now look at the specification of the process V in the module *Workcell-Behaviour*, which specifies the intended behaviour. From RDP it follows that a solution (v, e^n) exists. Now, if v is also a solution of the specification for W, RSP can be used to infer that v equals w.

Define k^n by:

$k^n = \tau . e^n . v$,

then we can derive

$v = \sum r1(n) . e^n . v = \sum r1(n) . k^n$

$k^0 = \tau . e^0 . v = \tau . s0(r) . v$

$k^{s(n)} = \tau . e^{s(n)} . v = \tau . s10(proc(p1)) . e^n . v$

So (v,k^n) is a solution of specification 2.

2.3. REDUNDANCY

Note that the specification of the workcell contains some redundancy. Although the transport service has the capability to store any number of products in the queue, this feature is not used in the workcell. At any moment not more than one product is stored in the buffer. So a one-item buffer would have functioned in the same way. Also, the option of receiving first a transport command and then a product is not used.

The capability of workstation A to receive a command to produce more than one product is also not used.

3. A WORKCELL WITH QUALITY CHECK

In this section a more complex workcell will be defined, having the possibility of checking the quality of the produced goods. The basic modules from the previous paragraphs are imported.

3.1. SPECIFICATION

3.1.1. Global Description

The workcell consists of four components:

WA	Workstation A. The workstation accepts a product, processes it and returns either a good product or a faulty product.
T	Transport service. A queue, at the one end accepting and at the other end sending products.
Q	Quality check. After receiving a product, the quality check determines whether it is a good product or not. A good product will be passed along, while a rejected product will be removed. The latter occurrence is signaled to the workcell controller.

WC Workcell Controller. This part controls the workcell. It receives the number of products that have to be processed, and instructs the workcell to do so. While the processing is going on, it will count the number of rejected products. At the end the workcell is instructed to process again an amount of products, equal to the number of rejections.

The workcell is graphically depicted in figure 3.

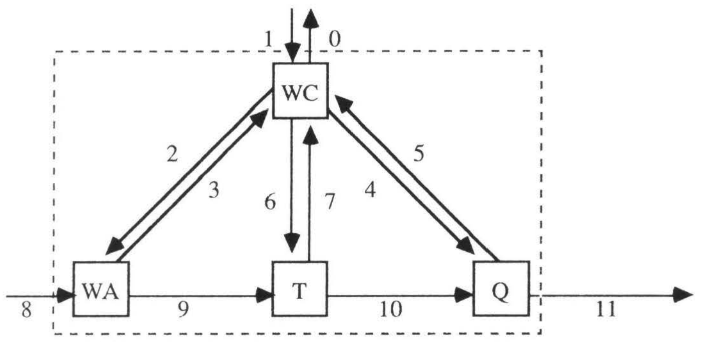


Figure 3 A workcell with quality check.

3.1.2. Datatypes Specific to the Workcell

The four components are connected to each other by 11 ports. The ports 0 through 7 are used to transmit data and the ports 8 through 11 are used to exchange products. The ports 0, 1, 8 and 11 are connected to the environment.

```
data module ports
begin

  exports
    begin
      sorts data-ports, product-ports
      functions
        port0  : -> data-ports
        port1  : -> data-ports
        port2  : -> data-ports
        port3  : -> data-ports
        port4  : -> data-ports
        port5  : -> data-ports
        port6  : -> data-ports
        port7  : -> data-ports
        port8  : -> product-ports
        port9  : -> product-ports
        port10 : -> product-ports
        port11 : -> product-ports
    end

end ports
```

The system has the set of products that can be produced or processed within the workstation as a parameter. The sort *products* contains at least product1 (*p1*) and the product *p1* after processing (*proc(p1)*). A quality (*ok* or *faulty*) can be added to a product using the function *prod*. This quality information can be revealed again by applying the function *qual*. These functions apply to normal products as well as to processed products. The sort of products which have a quality attached is called *qual-products*.

Note that the information about the quality of a product is attached to the product itself, and one can only become aware of it by explicitly using the *qual* funcion. As an example consider drilling a hole in some product. After drilling, the hole is in the right position or not, but one can only become aware of this after applying some measuring tool, which reveals the quality.

```
data module products
begin

  parameters
    products
      begin
        sorts products
        functions
          p1   :            -> products
          proc : products -> products
      end products

  exports
    begin
      sorts qual-products
      functions
        prod : products # quality -> qual-products
        qual : qual-products      -> quality
    end

  imports
    booleans
      {renamed by
        [BOOL  -> quality,
         true  -> ok,
         false -> fault]}

  variables
    p : -> products
    q : -> quality
  equations
  [1] qual(prod(p, q)) = q

end products
```

Along ports 1, 2, 4 and 6 a non-negative integer (*produce(n)*) can be sent to indicate that the receiver has to cope with *n* products. A ready message (*ready*) is sent back over the ports 0, 3, 5 and 7 to indicate that the component has fulfilled its task. Port 5 is also used to indicate that a product has been rejected (*reject*). These messages are declared in the following module.

```
data module messages
begin

  exports
    begin
      sorts messages
      functions
        produce : nat          -> messages
        ready   :              -> messages
        reject  :              -> messages
    end

  imports naturals

end messages
```

In the module *communication* all atomic actions, the communication function and the sets of atoms to be encapsulated and abstracted from are defined.

```
process module communication
begin

  exports
    begin
      atoms
        r : product-ports # qual-products
        s : product-ports # qual-products
        c : product-ports # qual-products
        r : data-ports # messages
        s : data-ports # messages
        c : data-ports # messages
      sets of data-ports
        internal-data-ports =
                  {port2, port3, port4, port5, port6, port7}
      of product-ports
        internal-product-ports = {port9, port10}
      of atoms
        I = { c(dp,m), c(pp,p) |
                dp in internal-data-ports, m in messages,
                pp in internal-product-ports, p in qual-products }
        H = { r(dp,m), s(dp,m), r(pp,p), s(pp,p) |
                dp in internal-data-ports, m in messages,
                pp in internal-product-ports, p in qual-products }
    end

  imports
    ports, products, messages

  communications
    r(dp,m) | s(dp,m) = c(dp,m)
      for dp in data-ports, m in messages
    r(pp,p) | s(pp,p) = c(pp,p)
      for pp in product-ports, p in qual-products

end communication
```

3.1.3. Workstation A

Workstation A is a machine able to process a specified number of products. This number is received over port 2. Then it executes its function *n* times (XA(n)). The process XA(0) simply sends a ready message via port 3 and starts the workstation all over. The process XA(s(n)) is able to receive some product which has to be processed. The possibility of either doing a good job or making an error while processing, is modeled by using the nondeterministic choice operator. By prefixing the actions with the internal atom *skip*, a choice is made which cannot be influenced by the environment.

```
process module Workstation-A
begin

  exports
    begin
      processes
        WA
    end

  imports
    communication, bounded-naturals

  processes
    XA : nat

  variables
    n : -> nat
  definitions
    WA        = sum(n in bounded-nat,
                  r(port2, produce(n)) . XA(n))
    XA(0)     = s(port3, ready) . WA
    XA(s(n))  = sum(p in products, r(port8, prod(p, ok)) .
                   (skip . s(port9, prod(proc(p), ok)) +
                    skip . s(port9, prod(proc(p), fault))) . XA(n))

end Workstation-A
```

3.1.4. Transport Service

The transport service can best be seen as a bounded FIFO-queue. First it receives the number of products that have to be transported. Then it behaves like the empty queue with bound *n*. After transporting *n* products (T(0, empty-queue)) a ready message is sent to the controller and it starts all over. The process T(n, q) is intended to model a queue with contents *q*, where *n* denotes the number of products that still have to be read in to the queue. T(s(n), empty-queue) has an empty buffer, so it can only read in products. T(0, add(r, q)) can only output the contents of its buffer. The process T(s(n), add(r, q)) can either accept some product or it can send a queued item. This transport service differs from the one defined in the previous section in the sense that it needs less external control and that the capability of buffering more than one product is being used. Also, its specification has less redundancy.

```
process module Transport-Service
begin

  exports
    begin
      processes
        T
    end

  imports
    communication, bounded-naturals,
    queues
       {items bound by
          [item -> qual-products]
      to products
       renamed by
          [queue -> product-queue]}

  processes
    T : nat # product-queue

  variables
    q : -> product-queue
    r : -> qual-products
    n : -> nat
  definitions
    T = sum(n in bounded-nat,
          r(port6, produce(n)) . T(n, empty-queue))
    T(0, empty-queue) = s(port7, ready) . T
    T(s(n), empty-queue) =
                sum(p in qual-products,
                  r(port9, p) . T(n, add-back(p, empty-queue)))
    T(0, add(r, q)) = s(port10, r) . T(0, q)
    T(s(n), add(r, q)) =
                sum(p in qual-products,
                  r(port9, p) . T(n, add-back(p, add(r, q)))) +
                s(port10, r) . T(s(n), q)

end Transport-Service
```

3.1.5. Quality Check

The quality of the processed product is tested by the process Q. It receives the command to test *n* products. Then the *n* tests are performed (XQ(n)). If there are no tests left to do (XQ(0)) a ready message is sent back and the quality check returns to its initial state. The checks are done by accepting some product *p* at port 10 and determining the quality of that product (XQ(n, p, qual(p))). If the quality is *ok* then the product can continue on its way. If the quality is *fault* then a rejection message is sent to the workcell controller and the product is rejected (i.e. discarded).

```
process module Quality-Check
begin

  exports
    begin
      processes
        Q
    end

  imports
    communication, bounded-naturals

  processes
    XQ : nat
    XQ : nat # qual-products # quality

  variables
    n : -> nat
    p : -> qual-products
  definitions
    Q                = sum(n in bounded-nat,
                         r(port4, produce(n)) . XQ(n))
    XQ(0)            = s(port5, ready) . Q
    XQ(s(n))         = sum(p in qual-products,
                         r(port10, p) .XQ(n, p, qual(p)))
    XQ(n, p, ok)     = s(port11, p) . XQ(n)
    XQ(n, p, fault)  = s(port5, reject) . XQ(n)

end Quality-Check
```

3.1.6. Workcell Controller

The workcell is controlled by the Workcell Controller. It receives the message to process *n* products. When this is done (D(0)), a ready message is reported to the environment and the controller starts all over. The process D(s(n)) handles the processing of *n+1* products. It sends the number of products that have to be processed to Workstation A, the Transport service and the Quality check. Then it starts to count the number of rejections, starting with 0 (RC(0)). The Rejection Counter will be incremented when it receives a rejection message. When the Quality check, the Transport service and Workstation A respectively send their ready messages, the controller again commands the workcell to process some number of products (D(n)). This new number of products is equal to the number of rejections encountered up to that moment.

```
process module Workcell-Controller
begin

  exports
    begin
      processes
        WC
    end

  imports
    communication, bounded-naturals
    processes
      D  : nat
      RC : nat

    variables
      n : -> nat
    definitions
      WC       = sum(n in bounded-nat,
                   r(port1, produce(n)) . D(n))
      D(0)     = s(port0, ready) . WC
      D(s(n))  = s(port4, produce(s(n))) . s(port6, produce(s(n))) .
                   s(port2, produce(s(n))) . RC(0)
      RC(n)    = r(port5, ready) . r(port7, ready). r(port3, ready) .
                   D(n) +
                 r(port5, reject) . RC(s(n))

  end Workcell-Controller
```

Note that the order in which the ready messages are received is of importance. If e.g. the ready message of WA can be received first, it is still possible for Q to contain faulty products. But then, since WC is not able to receive any rejection messages from Q, a deadlock would occur.

3.1.7. The Workcell

Now we are interested in the parallel operation of the four components as described above.

```
process module Workcell
begin

  exports
    begin
      processes
        W
    end

  imports
    Workstation-A, Quality-Check,
    Transport-Service, Workcell-Controller

  definitions
    W = hide(I, encaps(H, WC || T || WA || Q))

end Workcell
```

3.2. CORRECTNESS

3.2.1. Preliminaries

For the verification of this workcell we will need some more proof techniques., the Cluster Fair Abstraction Rule (CFAR) and the conditional axioms (see [4]).

The conditional axioms deal with distributing the encapsulation, hiding and merge operators.

$$
\boxed{\begin{array}{l}
\alpha(x) \mid (\alpha(y)\cap H) \subseteq H \;\Rightarrow\; \partial_H(x \parallel y) = \partial_H(x \parallel \partial_H(y)) \\
\alpha(x) \mid (\alpha(y)\cap I) = \varnothing \;\Rightarrow\; \tau_I(x \parallel y) = \tau_I(x \parallel \tau_I(y)) \\
\alpha(x)\cap H = \varnothing \;\Rightarrow\; \partial_H(x) = x \\
\alpha(x)\cap I = \varnothing \;\Rightarrow\; \tau_I(x) = x \\
H = H_1 \cup H_2 \;\Rightarrow\; \partial_H(x) = \partial_{H_1}\circ\partial_{H_2}(x) \\
I = I_1 \cup I_2 \;\Rightarrow\; \tau_I(x) = \tau_{I_1}\circ\tau_{I_2}(x) \\
H\cap I = \varnothing \;\Rightarrow\; \tau_I\circ\partial_H(x) = \partial_H\circ\tau_I(x)
\end{array}}
$$

table 1 Conditional Axioms

The $\alpha(x)$ operator determines the alphabet of a process, and is defined by

$$
\boxed{\begin{array}{l}
\alpha(\delta) = \varnothing \\
\alpha(\tau) = \varnothing \\
\alpha(a) = \{a\} \qquad (\text{if } a \neq \delta) \\
\alpha(\tau x) = \alpha(x) \\
\alpha(ax) = \{a\} \cup \alpha(x) \qquad (\text{if } a \neq \delta) \\
\alpha(x+y) = \alpha(x) \cup \alpha(y) \\
\alpha(x) = \bigcup_{n\geq 1} \alpha(\pi_n(x)) \\
\alpha(\tau_I(x)) = \alpha(x) - I
\end{array}}
$$

table 2 The Alphabet function

For the Cluster Fair Abstraction rule we need some definitions. Suppose *E* is a recursive specification over variables *V*, and suppose *I* is the set of atomic actions to be abstracted from. We call a subset C of *V* a cluster of *I* in *E* if for all *X* in C the equation for *X* in *E* has the form

$$X = \sum_{k=1}^{m} i_k . X_k + \sum_{l=1}^{n} Y_l,$$

where $m \geq 1$, $n \geq 0$, $i_1,\dots,i_m \in I \cup \{\tau\}$, $X_1,\dots,X_m \in C$, $Y_1,\dots,Y_n \in V-C$. The variables in C are called *cluster variables*. For variables $X,Y \in V$ we write $X \rightarrow Y$ if Y occurs in the right-hand side of the equation of X. Then, the exits of the cluster are those variables outside C, that can be reached from C, i.e.

$$\text{exits}(C) = \{Y \in V\text{-}C \mid X \rightarrow Y \text{ for some } X \in C\}.$$

Let $\rightarrow^*$ be the transitive and reflexive closure of $\rightarrow$. We call a cluster *C* of *I* in *E* conservative if every exit van be reached from every cluster variable, i.e. for all $X \in C$ and all $Y \in exits(C)$ we have $X \rightarrow^* Y$. Now we can formulate the rule CFAR as follows.

Let E be a guarded recursive specification; let $I \subseteq A$ be such that $|I| \geq 2$; let C be a finite conservative cluster of I in E; and let $X \in C$. Then

$$\tau_I(X) = \tau \cdot \sum_{Y \in \text{exits}(C)} \tau_I(Y)$$

3.2.2. Intended Behaviour

Now we have to define some criterion for correctness of the specification. It is not enough to require that for any command *produce(n)* along port 1 the workcell processes *n* products correctly and reports a ready message. The problem is that if there is not enough supply of products along port 8, the workcell can reach a deadlock situation, waiting for more products. So we will only consider the behaviour of the workcell in an environment, supplying an unlimited number of products. Thus we define the supplier *S*, which is repeatedly sending product *p1* along port 8.

Of course we have to encapsulate unsuccessful communications over port 8 and abstract from successful communications over this port.

```
process module System
begin

  exports
    begin
      processes
        W2
    end

  imports
    Workcell

  processes
    S

  sets of atoms
    I0 = { c(port8,p)  | p in qual-products }
    H0 = { r(port8,p), s(port8,p)  | p in qual-products }

  definitions
    S  = s(port8, prod(p1, ok)) . S
    W2 = hide(I0, encaps(H0, S || W))

end System
```

The extended configuration is depicted in figure 4.

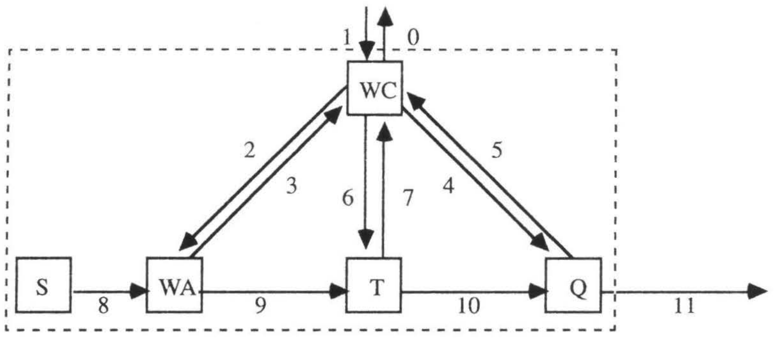


Figure 4 Adding a supplier to the workcell.

The intended behaviour can be specified by the following specification. A command to process *n* products correctly will be received, then the *n* processed products will be delivered and a ready message will be reported.

```
process module Workcell-Behaviour
begin

  exports
    begin
      processes
        V
    end

  imports
    communication, bounded-naturals

  processes
    E : nat

 variables
    n : -> nat
 definitions
    V        = sum(n in bounded-nat, r(port1, produce(n)) . E(n) . V)
    E(0)     = s(port0, ready)
    E(s(n)) = s(port11, prod(proc(p1), ok)) . E(n)

end Workcell-Behaviour
```

Now a verification of the correctness of the specification of the workcell will consist of a proof that the specification of *W2* and the specification of *V* define the same process. So if *w'* and *v* are solutions of the two specifications, we have to prove *v=w'*

3.2.3. The Workcell is Correct

THEOREM 2 *The specification of the workcell is correct.*

$ACP_\tau + RDP + RSP + ET + CFAR + CA \vdash v=w'$

PROOF
The proof consists of three steps. In step 1 we reduce the number of components, in step 2 we remove the parallelism and in step three we obtain the desired result by applying CFAR.

3.2.3.1. Step 1

First we reduce the number of components by aggregating the supplier S and Workstation A. The resulting process (K) can be seen as being a supplier of either good or bad processed products (See figure 5).

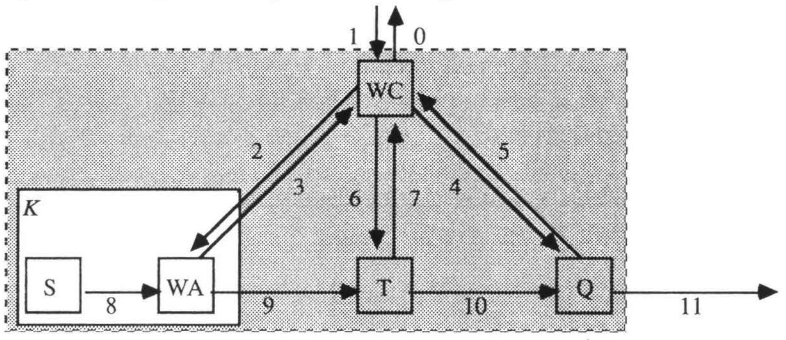


Figure 5 Aggregating S and WA.

Let the process K be specified by

$$
\begin{aligned}
K &= \Sigma r2(n) \,.\, XK^{n} \\
XK^{0} &= s3(r) \,.\, K \\
XK^{s(n)} &= (\tau \,.\, s9(prod(proc(p1), ok)) + \tau \,.\, s9(prod(proc(p1), fault))\,) \,.\, XK^{n}
\end{aligned}
$$

And let the encapsulation set and the abstraction set be defined by

$$H1 = \{rp(d), sp(d) \mid p=port8 \wedge d \in qual\text{-}products\}$$
$$I1 = \{cp(d) \mid p=port8 \wedge d \in qual\text{-}products\}$$

then the following proposition holds:

PROPOSITION

$$K = \tau_{I1}\, \partial_{H1}\, (S \,\|\, WA)$$

PROOF Let the process L be defined by

$$
\begin{aligned}
L &= \tau_{I1}\, \partial_{H1}\, (S \,\|\, WA) \\
&= \tau_{I1}\, \partial_{H1}\, (\, S \,\|\, \Sigma r2(n) \,.\, XA^{n}\,) \\
&= \Sigma r2(n) \,.\, \tau_{I1}\, \partial_{H1}\, (\, S \,\|\, XA^{n}\,)
\end{aligned}
$$

Let L^n be defined by

$L^n = \tau_{I1}\, \partial_{H1}$ (S $\|$ XA^n), then

$L^0 = \tau_{I1}\, \partial_{H1}$ (S $\|$ s3(r) . WA)
$= s3(r) . L$

$L^{s(n)} = \tau_{I1}\, \partial_{H1}$ (s8(prod(p1,ok)) . S $\|$
$\sum_{p\in \text{products}}$(r8(prod(p,ok)) . (skip . s9(prod(proc(p), ok)) +
skip . s9(prod(proc(p), fault)))) . XA^n)
$= \tau_{I1}$ (c8(prod(p1,ok)) .
∂_{H1} (S $\|$ (skip . s9(prod(proc(p1), ok)) +
skip . s9(prod(proc(p1), fault)))) . XA^n)
$= \tau$. (τ . s9(prod(proc(p1), ok)) + τ . s9(prod(proc(p1), fault)))) . L^n

Thus we have

$L = \sum r2(n) . L^n$
$L^0 = s3(r) . L$
$L^{s(n)} = \tau$. (τ . s9(prod(proc(p1), ok)) + τ . s9(prod(proc(p1), fault)))) . L^n

Now it is easy to see that K and L define the same process. Use RSP to prove that a solution of K is also a solution of system L.

As a consequence of this proposition we can replace the two components S and WA by one simpler component K. This technique is called local replacement and was introduced in [4]. In order to actually replace the two components in the specification of the workcell, we need the conditional axioms (see [8]).

W2 $= \tau_{I'}\, \partial_{H'}$ (S $\|$ W)
$= \tau_{I'}\, \partial_{H'}$ (S $\|$ $\tau_I\, \partial_H$ (WA $\|$ T $\|$ Q $\|$ WC))
$= \tau_{I'\cup I}\, \partial_{H'\cup H}$ (S $\|$ WA $\|$ T $\|$ Q $\|$ WC)
$= \tau_{I'\cup I}\, \partial_{H'\cup H}$ ($\tau_{I1}\, \partial_{H1}$(S $\|$ WA) $\|$ T $\|$ Q $\|$ WC)
$= \tau_{I'\cup I}\, \partial_{H'\cup H}$ (K $\|$ T $\|$ Q $\|$ WC)
$= \tau_I \partial_H$(K $\|$ T $\|$ Q $\|$ WC)

3.2.3.2. Step 2

In the second step we will remove the parallelism in the specification by expanding the merges. This will result in a complex process, which describes all states that the workcell has.

First we define a new abstraction set, I2, obtained by deleting the communication of the rejection message from the old one. This will be useful when applying CFAR in step 3.

$I2 = I \setminus \{c5(reject)\}$

If we define

$U \quad = \tau_{I2}\, \partial_H\, (K \| T \| Q \| WC)$, then we have

$W2 \quad = \tau_{\{c5(reject)\}}(U)$

For U we can derive

$$\begin{aligned} U &= \tau_{I2}\, \partial_H\, (K \| T \| Q \| WC) \\ &= \Sigma r1(n) \,.\, \tau_{I2}\, \partial_H\, (K \| T \| Q \| D^n) \end{aligned}$$

Let U^n be defined by $\tau_{I2}\, \partial_H\, (K \| T \| Q \| D^n)$, then

$$\begin{aligned} U^0 &= \tau_{I2}\, \partial_H\, (K \| T \| Q \| D^0) = s0(r) \,.\, U \\ U^{s(n)} &= \tau_{I2}\, \partial_H\, (K \| T \| Q \| D^{s(n)}) \\ &= \tau_{I2}\, (c4(s(n)) \,.\, c6(s(n)) \,.\, c2(s(n)) \,.\, \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\lambda} \| XQ^{s(n)} \| RC_0)) \end{aligned}$$

The process U^n denotes the total workcell, which has just received a command to produce a certain number of products. After distributing this command, the workcell enters the state in which the products will be produced. In the process of producing the products, there are several intermediate states. These states are determined by e.g. the number of products that still have to be produced, and the contents of the buffer of the transport service. The quality-check can also contain some product, i.e. the product which is read in and will be checked. All values that determine the actual state the workcell is in, are listed below:

choice	The choice made in K about processing correctly or faulty. The choice can be *ok* or *fault*. If no choice has been made yet, the value of this variable is $\times$
count	The number of products that still have to be produced (not considering the number of rejected products).
buffer	The contents of the buffer in the transport service. The value is λ if the buffer is empty.
Qcont	The contents of the quality-check part. The value is λ if Q contains no product.
rc	The rejection counter, counting the number of rejected products.

All states can be described using these five variables. Now it is possible to define the process *U*, indexed by these five variables, which describes the behaviour of the workcell during the production of the products.
Define

$U^{choice,\ count,\ buffer,\ Qcont,\ rc}$

as the composition of the four components K, T, Q and WC, where the indices determine the state of the four components as follows:

If $choice{=}\times$ then K is in state XK^{count}, otherwise K is in state $s9(prod(proc(p1),\ choice)).XK^{count-1}$.

T is in state $T_{count}{}^{buffer}$.

If $Qcont{=}\lambda$ then Q is in state $XQ^{count\ +\ |buffer|}$, otherwise Q is in state $XQ_{Qcont,\ qual(Qcont)}{}^{count\ +\ |buffer|}$.

WC is in state RC_{count}.

For every combination of values we can calculate the behaviour of the system. Note that the choice can only be unequal to $\times$ if the count is positive. Let *ch* be some quality (i.e. either *ok* or *fault*), let *n* and *rc* be natural numbers, let σ be a series of qual-products and let q be a qual-product.

$$
\begin{aligned}
&U^{ch,\ s(n),\ \sigma,\ prod(proc(p1),ok),\ rc}\\
&\quad = \tau_I\, \partial_H\, (s9(prod(proc(p1),ch)).XK^n \,\|\, T_{s(n)}{}^{\sigma} \,\|\, XQ_{p,ok}{}^{s(n)+|\sigma|} \,\|\, RC_{rc})\\
&\quad = \tau_I\, (c9(prod(proc(p1),ch)).\\
&\qquad\qquad \partial_H\, (XK^n \,\|\, T_n{}^{prod(proc(p1),ch)*\sigma} \,\|\, XQ_{p,ok}{}^{s(n)+|\sigma|} \,\|\, RC_{rc}) +\\
&\qquad\qquad s11(p).\ \partial_H\, (\, s9(prod(proc(p1),ch)).XK^n \,\|\, T_{s(n)}{}^{\sigma} \,\|\, XQ^{s(n)+|\sigma|} \,\|\, RC_{rc})\,)\\
&\quad = \tau\,.\ U^{\times,\ n,\ prod(proc(p1),ch)*\sigma,\ prod(proc(p1),ok),\ rc} +\\
&\qquad s11(prod(proc(p1),ok))\,.\ U^{ch,\ s(n),\ \sigma,\ \lambda,\ rc}\\
&U^{ch,\ s(n),\ \sigma,\ prod(proc(p1),fault),\ rc}\\
&\quad = \tau_I\, \partial_H\, (s9(prod(proc(p1),ch)).XK^n \,\|\, T_{s(n)}{}^{\sigma} \,\|\, XQ_{p,fault}{}^{s(n)+|\sigma|} \,\|\, RC_{rc})\\
&\quad = \tau_I\, (c9(prod(proc(p1),ch)).\\
&\qquad\qquad \partial_H\, (XK^n \,\|\, T_n{}^{prod(proc(p1),ch)*\sigma} \,\|\, XQ_{p,fault}{}^{s(n)+|\sigma|} \,\|\, RC_{rc}) +\\
&\qquad\qquad c5(rej).\ \partial_H\, (\, s9(prod(proc(p1),ch)).XK^n \,\|\, T_{s(n)}{}^{\sigma} \,\|\, XQ^{s(n)+|\sigma|} \,\|\, RC_{s(rc)})\,)\\
&\quad = \tau\,.\ U^{\times,\ n,\ prod(proc(p1),ch)*\sigma,\ prod(proc(p1),fault),\ rc} +\\
&\qquad c5(rej)\,.\ U^{ch,\ s(n),\ \sigma,\ \lambda,\ s(rc)}\\
&U^{ch,\ s(n),\ \sigma*q,\ \lambda,\ rc}\\
&\quad = \tau_I\, \partial_H\, (s9(prod(proc(p1),ch)).XK^n \,\|\, T_{s(n)}{}^{\sigma*q} \,\|\, XQ^{n+2+|\sigma|} \,\|\, RC_{rc})\\
&\quad = \tau_I\, (c9(prod(proc(p1),ch)).\\
&\qquad\qquad \partial_H\, (XK^n \,\|\, T_n{}^{prod(proc(p1),ch)*\sigma*q} \,\|\, XQ^{n+2+|\sigma|} \,\|\, RC_{rc}) +\\
&\qquad\qquad c10(q).\ \partial_H\, (\, s9(prod(proc(p1),ch)).XK^n \,\|\, T_{s(n)}{}^{\sigma} \,\|\\
&\qquad\qquad\qquad\qquad XQ_{q,qual(q)}{}^{s(n)+|\sigma|} \,\|\, RC_{rc})\,)\\
&\quad = \tau\,.\ U^{\times,\ n,\ prod(proc(p1),ch)*\sigma*q,\ \lambda,\ rc} +
\end{aligned}
$$

$\tau \,.\, U^{ch,\, s(n),\, \sigma,\, q,\, rc}$

$U^{ch,\, s(n),\, \lambda,\, \lambda,\, rc}$

$= \tau_I\, \partial_H\, (s9(prod(proc(p1),ch)).XK^n \| T_{s(n)}{}^{\lambda} \| XQ^{s(n)} \| RC_{rc})$

$= \tau_I\, (c9(prod(proc(p1),ch)).\ \partial_H\, (XK^n \| T_n{}^{prod(proc(p1),ch)} \| XQ^{s(n)} \| RC_{rc})\,)$

$= \tau \,.\, U^{\times,\, n,\, prod(proc(p1),ch),\, \lambda,\, rc}$

$U^{\times,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc}$

$= \tau_I\, \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\sigma} \| XQ_{p,ok}{}^{s(n)+|\sigma|} \| RC_{rc})$

$= \tau_I\, (\ \tau.\ \partial_H\, (\ s9(prod(proc(p1),ok))\ XK^n \| T_{s(n)}{}^{\sigma} \| XQ_{p,ok}{}^{s(n)+|\sigma|} \| RC_{rc}\,) +$

$\tau.\ \partial_H\, (\ s9(prod(proc(p1),fault))\ XK^n \| T_{s(n)}{}^{\sigma} \| XQ_{p,ok}{}^{s(n)+|\sigma|} \| RC_{rc}\,) +$

$s11(p).\ \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\sigma} \| XQ^{s(n)+|\sigma|} \| RC_{rc})\,)$

$= \tau \,.\, U^{ok,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc}\ +$

$\tau \,.\, U^{fault,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc}\ +$

$s11(prod(proc(p1),ok))\ .\ U^{\times,\, s(n),\, \sigma,\, \lambda,\, rc}$

$U^{\times,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc}$

$= \tau_I\, \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\sigma} \| XQ_{p,fault}{}^{s(n)+|\sigma|} \| RC_{rc})$

$= \tau_I\, ($

$\tau.\ \partial_H\, (\ s9(prod(proc(p1),ok))\ XK^n \| T_{s(n)}{}^{\sigma} \| XQ_{p,fault}{}^{s(n)+|\sigma|} \| RC_{rc}\,) +$

$\tau.\ \partial_H\, (\ s9(prod(proc(p1),fault))\ XK^n \| T_{s(n)}{}^{\sigma} \| XQ_{p,fault}{}^{s(n)+|\sigma|} \| RC_{rc}\,) +$

$c5(rej).\ \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\sigma} \| XQ^{s(n)+|\sigma|} \| RC_{s(rc)})\,)$

$= \tau \,.\, U^{ok,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc}\ +$

$\tau \,.\, U^{fault,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc}\ +$

$c5(rej)\ .\ U^{\times,\, s(n),\, \sigma,\, \lambda,\, s(rc)}$

$U^{\times,\, s(n),\, \sigma^*q,\, \lambda,\, rc}$

$= \tau_I\, \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\sigma^*q} \| XQ^{n+2+|\sigma|} \| RC_{rc})$

$= \tau_I\, (\ \tau.\ \partial_H\, (\ s9(prod(proc(p1),ok))\ XK^n \| T_{s(n)}{}^{\sigma^*q} \| XQ^{n+2+|\sigma|} \| RC_{rc}\,) +$

$\tau.\ \partial_H\, (\ s9(prod(proc(p1),fault))\ XK^n \| T_{s(n)}{}^{\sigma^*q} \| XQ^{n+2+|\sigma|} \| RC_{rc}\,) +$

$c10(q).\ \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\sigma} \| XQ_{q,qual(q)}{}^{s(n)+|\sigma|} \| RC_{rc})\,)$

$= \tau \,.\, U^{ok,\, s(n),\, \sigma^*q,\, \lambda,\, rc} + \tau \,.\, U^{fault,\, s(n),\, \sigma^*q,\, \lambda,\, rc} + \tau \,.\, U^{\times,\, s(n),\, \sigma,\, q,\, rc}$

$U^{\times,\, s(n),\, \lambda,\, \lambda,\, rc}$

$= \tau_I\, \partial_H\, (XK^{s(n)} \| T_{s(n)}{}^{\lambda} \| XQ^{s(n)} \| RC_{rc})$

$= \tau_I\, (\ \tau.\ \partial_H\, (\ s9(prod(proc(p1),ok))\ XK^n \| T_{s(n)}{}^{\lambda} \| XQ^{s(n)} \| RC_{rc}\,) +$

$\tau.\ \partial_H\, (\ s9(prod(proc(p1),fault))\ XK^n \| T_{s(n)}{}^{\lambda} \| XQ^{s(n)} \| RC_{rc}\,)$

$= \tau \,.\, U^{ok,\, s(n),\, \lambda,\, \lambda,\, rc} + \tau \,.\, U^{fault,\, s(n),\, \lambda,\, \lambda,\, rc}$

$U^{\times,\, 0,\, \sigma,\, prod(proc(p1),ok),\, rc}$

$= \tau_I\, \partial_H\, (XK^0 \,\|\, T_0^{\sigma} \,\|\, XQ_{p,ok}{}^{|\sigma|} \,\|\, RC_{rc})$

$= \tau_I\, (\, s11(p).\, \partial_H\, (\, XK^0 \,\|\, T_0^{\sigma} \,\|\, XQ^{|\sigma|} \,\|\, RC_{rc}\,)\,)$

$= s11(prod(proc(p1),ok)).\ U^{\times,\, 0,\, \sigma,\, \lambda,\, rc}$

$U^{\times,\, 0,\, \sigma,\, prod(proc(p1),fault),\, rc}$

$= \tau_I\, \partial_H\, (XK^0 \,\|\, T_0^{\sigma} \,\|\, XQ_{p,fault}{}^{|\sigma|} \,\|\, RC_{rc})$

$= \tau_I\, (\, c5(rej).\, \partial_H\, (\, XK^0 \,\|\, T_0^{\sigma} \,\|\, XQ^{|\sigma|} \,\|\, RC_{s(rc)}\,)\,)$

$= c5(rej) \,.\, U^{\times,\, 0,\, \sigma,\, \lambda,\, s(rc)}$

$U^{\times,\, 0,\, \sigma^*q,\, \lambda,\, rc}$

$= \tau_I\, \partial_H\, (XK^0 \,\|\, T_0^{\sigma^*q} \,\|\, XQ^{|\sigma|+1} \,\|\, RC_{rc})$

$= \tau_I\, (\, c10(q).\, \partial_H\, (\, XK^0 \,\|\, T_0^{\sigma} \,\|\, XQ_{q,qual(q)}{}^{|\sigma|} \,\|\, RC_{rc}\,)\,)$

$= \tau.\ U^{\times,\, 0,\, \sigma,\, q,\, rc}$

$U^{\times,\, 0,\, \lambda,\, \lambda,\, rc}$

$= \tau_I\, \partial_H\, (XK^0 \,\|\, T_0^{\lambda} \,\|\, XQ^0 \,\|\, RC_{rc})$

$= \tau_I\, (\, c5(r).c7(r).c3(r).\, \partial_H\, (\, K \,\|\, T \,\|\, Q \,\|\, D^{rc}\,)\,)$

$= \tau.\ U^{rc}$

Thus we have the following system:

1) $U = \Sigma r1(n) \,.\, U^n$

2) $U^0 = s0(r) \,.\, U$

3) $U^{s(n)} = \tau \,.\, U^{\times,\, s(n),\, \lambda,\, \lambda,\, 0}$

4) $U^{ch,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc} =$

$\tau \,.\, U^{\times,\, n,\, prod(proc(p1),ch)^*\sigma,\, prod(proc(p1),ok),\, rc} +$

$s11(prod(proc(p1),ok)) \,.\, U^{ch,\, s(n),\, \sigma,\, \lambda,\, rc}$

5) $U^{ch,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc} =$

$\tau \,.\, U^{\times,\, n,\, prod(proc(p1),ch)^*\sigma,\, prod(proc(p1),fault),\, rc} +$

$c5(rej) \,.\, U^{ch,\, s(n),\, \sigma,\, \lambda,\, s(rc)}$

6) $U^{ch,\, s(n),\, \sigma^*q,\, \lambda,\, rc} =$

$\tau \,.\, U^{\times,\, n,\, prod(proc(p1),ch)^*\sigma^*q,\, \lambda,\, rc} +$

$\tau \,.\, U^{ch,\, s(n),\, \sigma,\, q,\, rc}$

7) $U^{ch,\, s(n),\, \lambda,\, \lambda,\, rc} = \tau \,.\, U^{\times,\, n,\, prod(proc(p1),ch),\, \lambda,\, rc}$

8) $U^{\times,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc} =$

$\tau \,.\, U^{ok,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc}\ +$

$\tau \,.\, U^{fault,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc}\ +$

$s11(prod(proc(p1),ok)) \,.\, U^{\times,\, s(n),\, \sigma,\, \lambda,\, rc}$

9) $U^{\times,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc} =$

$\tau \,.\, U^{ok,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc}\ +$

$\tau \,.\, U^{fault,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc}\ +$

$c5(rej) \,.\, U^{\times,\, s(n),\, \sigma,\, \lambda,\, s(rc)}$

10) $U^{\times,\, s(n),\, \sigma^*q,\, \lambda,\, rc} =$

$\tau \,.\, U^{ok,\, s(n),\, \sigma^*q,\, \lambda,\, rc}\ +$

$\tau \,.\, U^{fault,\, s(n),\, \sigma^*q,\, \lambda,\, rc}\ +$

$\tau \,.\, U^{\times,\, s(n),\, \sigma,\, q,\, rc}$

11) $U^{\times,\, s(n),\, \lambda,\, \lambda,\, rc} = \tau \,.\, U^{ok,\, s(n),\, \lambda,\, \lambda,\, rc} + \tau \,.\, U^{fault,\, s(n),\, \lambda,\, \lambda,\, rc}$

12) $U^{\times,\, 0,\, \sigma,\, prod(proc(p1),ok),\, rc} = s11(prod(proc(p1),ok)).\, U^{\times,\, 0,\, \sigma,\, \lambda,\, rc}$

13) $U^{\times,\, 0,\, \sigma,\, prod(proc(p1),fault),\, rc} = c5(rej) \,.\, U^{\times,\, 0,\, \sigma,\, \lambda,\, s(rc)}$

14) $U^{\times,\, 0,\, \sigma^*q,\, \lambda,\, rc} = \tau.\, U^{\times,\, 0,\, \sigma,\, q,\, rc}$

15) $U^{\times,\, 0,\, \lambda,\, \lambda,\, rc} = \tau.\, U^{rc}$

specification 2

3.2.3.3. Step 3

In the final part of the proof we use CFAR (see [4]) and RSP to prove that the system derived in step 2 can be reduced to the desired specification *V*.

Some observations about the specification above can be made. The number of products that still have to be produced correctly (*m*) can be determined from the values of the indices of the process:

$m = count + |buffer| + |Qcont| + rc$

So we must prove the equality

$\tau.\, E^m = \tau.\tau_{\{c5(reject)\}}(U^{choice,\, count,\, buffer,\, Qcont,\, rc})$

We must also prove

$\tau.E^m = \tau.\tau_{\{c5(reject)\}}(U^m).$

Comparing the two processes one easily notes that U^m has the possibility to produce only faulty products, hence it can loop forever, sending rejection messages. The process E^m however does not have this possibility. Thus we must make the assumption that workstation *WA* is not completely broken. It now and then must process some product correctly. This fairness assumption can be modeled in process algebra with the Cluster Fair Abstraction Rule.

The only cases in which it is possible to never process a product correctly are the processes which are indexed such that (i) *choice≠ok*, (ii) the buffer contains no correctly processed products and (iii) *Qcont≠prod(proc(p1),ok)*. This observation leads us to consider clusters of processes which satisfy these conditions and have to produce the same number of products. Thus cluster *m* (for *m*>0) is defined by:

$$CL(m) = \{U^m\} \cup \{ U^{choice,\ count,\ buffer,\ Qcont,\ rc} \mid choice \neq ok \wedge prod(proc(p1),ok) \notin buffer \wedge Qcont \neq prod(proc(p1),ok) \wedge count + |buffer| + |Qcont| + rc = m\}$$

This defines a conservative cluster from *{c5(reject)}* in the specification of $U^{choice,\ count,\ buffer,\ Qcont,\ rc}$ (using terminology of [4]). The Workcell can choose to loop forever in such a cluster, or it can choose to process some product correctly. This will be indicated by setting the *choice*-index to *ok*. After some time, this choice leads to a correctly processed product leaving the workcell. In the meantime the workcell has to make new choices. If they are all negative, we again enter a cluster that permits infinite loops. If a choice was made to produce one or more correct products, we are still in a state in which progress can be made.

Now we can determine the exits of such a cluster. These are all states which can be reached from the cluster, but are no member of it. Thus there are no correctly processed products in the buffers and the choice has been made to process the next product correctly.

$$\begin{aligned} EXITS(m) = \ & \{U^{ok,\ s(n),\ \sigma,\ prod(proc(p1),fault),rc} \mid \\ & \qquad s(n) + |\sigma| + 1 + rc = m \wedge \pi\rho o\delta(proc(p1),ok) \notin \sigma \} \cup \\ & \{U^{ok,\ s(n),\ \sigma*q,\ \lambda,\ rc} \mid \\ & \qquad s(n) + |\sigma| + 1 + rc = m \wedge prod(proc(p1),ok) \notin \sigma*q \} \cup \\ & \{U^{ok,\ s(n),\ \lambda,\ \lambda,\ rc} \mid s(n) + rc = m\} \end{aligned}$$

Applying CFAR to the specification derived in step 2 leads to a new specification. This specification is equal to the old one for states which contain some correctly processed product and is modified for states which only contain faulty products.

Now set

$$W' = \tau_{\{c5(reject)\}}(U)$$

$$W^n = \tau_{\{c5(reject)\}}(U^n)$$

$$W^{choice,\ count,\ buffer,\ Qcont,\ rc} = \tau_{\{c5(reject)\}}(U^{choice,\ count,\ buffer,\ Qcont,\ rc})$$

In the first part of the following specification we assume that there are correctly processed products in the buffer σ, or in *Qcont*, or *ch=ok*. The numbers correspond to the numbers in the specification of *U*.

1) $W' = \Sigma_{n\geq 0} r1(n) \, . \, W^{n}$

2) $W^{0} = s0(r) \, . \, W'$

4) $W^{ch,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc} =$

$\quad \tau \, . \, W^{\times,\, n,\, prod(proc(p1),ch)*\sigma,\, prod(proc(p1),ok),\, rc} +$

$\quad s11(prod(proc(p1),ok)) \, . \, W^{ch,\, s(n),\, \sigma,\, \lambda,\, rc}$

5) $W^{ch,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc} =$

$\quad \tau \, . \, W^{\times,\, n,\, prod(proc(p1),ch)*\sigma,\, prod(proc(p1),fault),\, rc} +$

$\quad \tau \, . \, W^{ch,\, s(n),\, \sigma,\, \lambda,\, s(rc)}$

6) $W^{ch,\, s(n),\, \sigma*q,\, \lambda,\, rc} =$

$\quad \tau \, . \, W^{\times,\, n,\, prod(proc(p1),ch)*\sigma*q,\, \lambda,\, rc} +$

$\quad \tau \, . \, W^{ch,\, s(n),\, \sigma,\, q,\, rc}$

7) $W^{ch,\, s(n),\, \lambda,\, \lambda,\, rc} = \tau \, . \, W^{\times,\, n,\, prod(proc(p1),ch),\, \lambda,\, rc}$

8) $W^{\times,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc} =$

$\quad \tau \, . \, W^{ok,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc} +$

$\quad \tau \, . \, W^{fault,\, s(n),\, \sigma,\, prod(proc(p1),ok),\, rc} +$

$\quad s11(prod(proc(p1),ok)) \, . \, W^{\times,\, s(n),\, \sigma,\, \lambda,\, rc}$

9) $W^{\times,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc} =$

$\quad \tau \, . \, W^{ok,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc} +$

$\quad \tau \, . \, W^{fault,\, s(n),\, \sigma,\, prod(proc(p1),fault),\, rc} +$

$\quad \tau \, . \, W^{\times,\, s(n),\, \sigma,\, \lambda,\, s(rc)}$

10) $W^{\times,\, s(n),\, \sigma*q,\, \lambda,\, rc} =$

$\quad \tau \, . \, W^{ok,\, s(n),\, \sigma*q,\, \lambda,\, rc} +$

$\quad \tau \, . \, W^{fault,\, s(n),\, \sigma*q,\, \lambda,\, rc} +$

$\quad \tau \, . \, W^{\times,\, s(n),\, \sigma,\, q,\, rc}$

12) $W^{\times,\, 0,\, \sigma,\, prod(proc(p1),ok),\, rc} = s11(prod(proc(p1),ok)). \, W^{\times,\, 0,\, \sigma,\, \lambda,\, rc}$

13) $W^{\times,\, 0,\, \sigma,\, prod(proc(p1),fault),\, rc} = \tau \, . \, W^{\times,\, 0,\, \sigma,\, \lambda,\, s(rc)}$

14) $W^{\times,\, 0,\, \sigma*q,\, \lambda,\, rc} = \tau. \, W^{\times,\, 0,\, \sigma,\, q,\, rc}$

specification 3 part 1

In the second part we assume that there are no correct products in the workcell, so we are in a cluster. The expression Σ EXITS(m) is shorthand for

$\quad \Sigma_{\, p \in EXITS(m)} \, \tau_{\{c5(reject)\}}(p)$.

3) $W^{s(n)} = \tau \,.\, \sum \text{EXITS}(s(n))$

5a) $W^{ch,\, s(n),\, \sigma,\, proc(p1,fault),\, rc} = \tau \,.\, \sum \text{EXITS}(s(n)+|\sigma|+1+rc)$

6a) $W^{ch,\, s(n),\, \sigma{*}q,\, \lambda,\, rc} = \tau \,.\, \sum \text{EXITS}(s(n)+|\sigma|+1+rc)$

7a) $W^{ch,\, s(n),\, \lambda,\, \lambda,\, rc} = \tau \,.\, \sum \text{EXITS}(s(n)+rc)$

9a) $W^{\times,\, s(n),\, \sigma,\, proc(p1,fault),\, rc} = \tau \,.\, \sum \text{EXITS}(s(n)+|\sigma|+1+rc)$

10a) $W^{\times,\, s(n),\, \sigma{*}q,\, \lambda,\, rc} = \tau \,.\, \sum \text{EXITS}(s(n)+|\sigma|+1+rc)$

11) $W^{\times,\, s(n),\, \lambda,\, \lambda,\, rc} = \tau \,.\, \sum \text{EXITS}(s(n)+rc)$

13a) $W^{\times,\, 0,\, \sigma,\, proc(p1,fault),\, rc} = \tau \,.\, \sum \text{EXITS}(|\sigma|+1+rc)$

14a) $W^{\times,\, 0,\, \sigma{*}q,\, \lambda,\, rc} = \tau \,.\, \sum \text{EXITS}(|\sigma|+1+rc)$

15) $W^{\times,\, 0,\, \lambda,\, \lambda,\, rc} = \tau \,.\, \sum \text{EXITS}(rc)$

specification 3 part 2

This specification now describes exactly the same process as the specification of V from the module *Workvcell-Behaviour*. This can be easily verified by substituting V for W', $E^0.V$ for W^0, $\tau.E^{s(n)}$ for $W^{s(n)}$ and the process $\tau.E^{count+|buffer|+|Qcont|+rc}$ for $W^{ch,count,buffer,Qcont,rc}$. Note that the only equation not starting with a τ is equation 12. So we must substitute $E^{|\sigma|+1}$ for $W^{\times,0,\sigma,proc(p1,ok)}$. So we see that V is a solution of the system defining W', and thus we can use RSP to conclude that v equals w'.

Note that RSP is only applicable if the specifications are guarded. A proof of the guardedness of specification 3 is straightforward.

4. Final Remarks

The techniques introduced in this chapter seem to be powerful enough to aid in the specification and verification of CIM-architectures. Although two workcells were considered of low complexity, the basic concepts of the technique are well illustrated. Now, due to the compositionality of the specifications, one can build a large plant consisting of a number of workcells which are already proved to function correctly. Thus, increasing the scale of the system will be possible.

It is also possible to add new features to the workcell and model them in process algebra. Possible features are: interrupts (modeled by the priority-operator, see [7]), detailed reports on the functioning of a machine, changing the tools of a machine, etc. Most of these features are not more complex than adding quality checks to a workcell.

Since a wide range of proof-rules and proof-techniques are developed in process algebra, the specification of a CIM-architecture in process algebra has advantages over specification in e.g. LOTOS. To name one, in LOTOS there is no equivalent of the fairness assumption.

Chapter 6

SPECIFICATION OF THE TRANSIT NODE IN PSF

(with F. Wiedijk)

The specification language PSF is used to give a formal specification of a transit node, a common case study in ESPRIT project METEOR. The design of the specification derived from the informal text and the ERAE specification is included. A short discussion on the relation to the specification in ERAE is provided.

1. INTRODUCTION

This chapter contains a case study in the formal description technique PSF. We specify a transit node, which is the common case study for several formalisms in the ESPRIT project nr. 432, METEOR. The PSF specification is derived partially from an informal text and partially from the ERAE specification in [54]. The design of the specification is included, from which a general method can be derived for specifying similar problems in PSF. In [80] the transit node is specified in the algebraic specification language PLUSS.

The PSF specification can be viewed as a more implementation directed specification than the one in ERAE. Certain design decisions are made, for example in identifying the separate objects that act in parallel. Thus the PSF specification, viewed as an implementation of the ERAE specification, must be verified or validated. A short discussion is devoted to this topic.

2. THE TRANSIT NODE

The Transit Node is a case study, which was defined in the RACE project 1046 (SPECS). An informal description of the Transit Node and the ERAE specification of it can be found in [54]. The informal specification reads as follows:

"The system to be specified consists of a transit node with:

- *1 Control Port-In*
- *1 Control Port-Out*
- *N Data Ports-In*
- *N Data Ports-Out*
- *M Routes Through*

(The limits of N and M are not specified.)

Each port is serialized. All ports are concurrent to all others. The ports should be specified as separate, concurrent entities. Messages arrive from the environment only when a Port-In is abe to treat them.

The node is "fair". All messages are equally likely to be treated, when a selection must be made, and all messages will eventually transit the node, or be placed in the collection of faulty messages.

Initial State: 1 Control Port-In, 1 Control Port-Out.

The Control Port-In accepts and treats the following three messages:

- *Add-Data-Port-In-&-Out(n)*

gives the node knowledge of a new port-in(n) and a new port-out(n). The node commences to accept and treat messages sent to the port-in, as indicated below on Data Port-In.

- *Add-Route((m),n(i),n(j),...))*

gives the node knowledge of a route associating route m with Data Port-Out(n(i),n(j),...).

- *Send-Faults*

routes all saved faulty messages, if any to Control-Port-Out. The order in which the faulty messages are transmitted is not specified.

A Data Port-In accepts and treats only messages of the type:

- *Route(m).Data*

The Port-In routes the message, unchanged, to any one (non-determinate) of the Data Ports-Out associated with route m. (Note that a Data Port-Out is serialized - the message has to be buffered until the Data Port-Out can process it). The message becomes a faulty message if *its transit time through the node (from initial receipt by a Data Port-In to transmission by a Data Port-Out) is greater than a constant time T.*

Data Ports-Out and Control Port-Out accept messages of any type and will transmit the message out of the node. Messages may leave the node in any order.

All faulty messages are saved until a Send-Faults command message causes them to be routed to Control Port-Out. Faulty messages are messages on the Control Port-In that are not one of the three commands listed, messages on a Data Port-In that indicate an unknown route, or messages whose transit time through the node is greater than T. Messages that exceed the transit time of T become faulty as soon as the time T is exceeded. It is permissible for a faulty message to not be routed to Control Port-Out (because, for example, it has just become faulty, but has not yet been placed in a faulty message collection), but all faulty messages must eventually be sent to Control Port-Out with a succession of Send-Faults commands.

It may be assumed that a source of time (time-of-day or a signal each time interval) is available in the environment and need not be modeled with the specification."

3. Design of the Specification

3.1. General

The specification was designed using a mixed top-down and bottom-up approach. It was based on the informal text, while using the interpretation of the text in the ERAE specification when needed to fill in omissions or solve ambiguities.

Several design decisions were made, which did not follow directly from the informal description of the case study (for example, the decision to let the Control Port-in keep control of the table containing all routes through the node).

3.2. Design

We first identify all parameters of the system, that are objects which are -and should be- unspecified. Since *"it may be assumed that a source of time is available in the environment"*, we postulate the existence of a process that behaves like a *clock*. This can be done by specifying a parameter containing this *clock* process. The second parameter is formed by the time that a message may be inside the node without getting faulty, the *maximal transit time*. The exact length of this duration should be decided upon at the implementation phase.

Then we identify all (concurrent) components in the system. We have a *Control-Port-In*, a *Control-Port-Out*, a number of *Data-Ports-in* and a number of *Data-Ports-Out*. Note that we will not consider the *Routes* as components, since these are static objects without temporal behaviour. Because all *Data-Ports-In* have the same behaviour, we can specify just one process, indexed with the actual name of the port. The same holds for the *Data-Ports Out*.

Now we make the decision that the routes and the information about the ports that exist are handled by the *Control-Port-In*, so this process is indexed with a *route-table* and with a *port-set*. Furthermore we see that the *Control-Port-Out* must contain a number of faulty messages that should be flushed and that every *Data-Port-Out* must contain a number of messages that should be sent to the environment. So both processes are indexed with a *message-bag*. The signature of the top-level objects now looks like:

```
processes
  control-port-in : route-table # port-set
  control-port-out : message-bag
  data-port-in : port-name
  data-port-out : port-name # message-bag
```

From the informal text and the ERAE specification we can now define the initial state of the node. It consists of the concurrent operation of the *control-port-in* and the *control-port-out*, indexed with the *empty-route-table*, the *empty-port-set* and the *empty-message-bag*. Of course we must add the parameter process *clock* in parallel and we must abstract from the internal actions and encapsulate unsuccessful communications.

```
transit-node = hide(I, encaps(H,
      clock ||
      control-port-in(empty-route-table, empty-port-set) ||
      control-port-out(empty-message-bag)))
```

Now we can proceed in a bottom up way by defining the data types *route-table* (an instance of the parameterized module *table* with the data type *routes*), *port-set* (*sets* instantiated with *ports*), *message-bag* (*bags* instantiated with *messages*) and *port-name*.

The top-down approach is continued by defining the behaviour of the four processes, each in a separate module. This leads to the question which objects are connected, in order to communicate to each other. We see that there is a link between the *control-port-in* and the *control-port-out*. Every *data-port-in* is linked to the *control-port-in* for route information and to the *control-port-out* for sending faulty messages. All *data-ports-in* are connected to all *data-ports-out* to transmit messages. And finally all ports have a connection to the environment for either accepting or transmitting messages.

As can be seen in the specification, the behaviour of the objects is specified by determining all initial communication actions. Every action is then followed by the corresponding behaviour, such as a transmission or a state change. This can possibly be specified by using subprocesses.

The *control-port-in* e.g. can accept one of the following messages:

- *add-datum-port(p),* followed by the subprocess that handles adding a *data-port-in* and a *data-port-out;*
- *add-route(r),* followed by a state change where the *route-table* is updated;
- *send-faults,* followed by forwarding this message to *control-port-out;*
- *request-route(rn),* followed by sending appropriate information about the route back.

After having identified all atomic actions (i.e. communication attempts) we can define the communication function and the set of atoms that has to be encapsulated and abstracted.

3.3. TOPOLOGY OF THE TRANSIT NODE

We can visualize the structure of the transit node with the following picture.

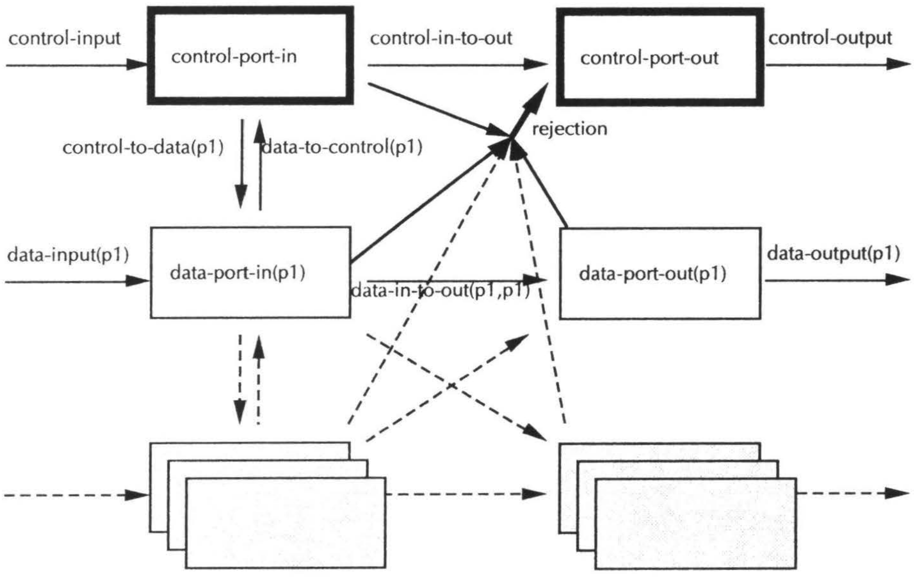


figure 1 The transit node

4. THE SPECIFICATION

The specification that resulted from the design as described in the previous paragraph will now be given. Note that the linear structure of the specification does not comply with the way the specification was designed. This is because the formalism forces us to write down the specification in a bottom-up way.

We first give all basic data types needed in the specification, then we define the data types specific to the transit node, then we define all processes involved and finally we give an example of an instantiation of the clock parameter.

4.1. BASIC DATA TYPES

The basic data types consist of the simple types *booleans* and *natural numbers*, and the parameterized types *bags, sets* and *tables*. The difference between bags and sets is that in a set duplicates are removed. A table can be used to look up an item corresponding to the value of a certain key.

```
data module booleans
begin

   exports
      begin
         sorts BOOL
         functions
           true  :               -> BOOL
           false :               -> BOOL
           or    : BOOL # BOOL -> BOOL
           and   : BOOL # BOOL -> BOOL
      end

   variables
      b : -> BOOL
   equations
      [1]  or(true, b)    = true
      [2]  or(false, b)   = b
      [3]  and(true, b)   = b
      [4]  and(false, b)  = false

end booleans
```

```
data module natural-numbers
begin

   exports
      begin
         sorts nat
         functions
           0     :             -> nat
           s     : nat         -> nat
           eq    : nat # nat -> BOOL
           lt    : nat # nat -> BOOL
           _ + _ : nat # nat -> nat
           _ - _ : nat # nat -> nat
      end

   imports booleans

   variables
      n, n1, n2 : -> nat
   equations
      [1]  eq(0, 0)            = true
      [2]  eq(0, s(n))         = false
      [3]  eq(s(n), 0)         = false
      [4]  eq(s(n1), s(n2))    = eq(n1, n2)
      [5]  lt(0, s(n))         = true
      [6]  lt(n, 0)            = false
      [7]  lt(s(n1), s(n2))    = lt(n1, n2)
      [8]  n + 0               = n
      [9]  n1 + s(n2)          = s(n1 + n2)
     [10]  0 - n               = 0
     [11]  n - 0               = n
     [12]  s(n1) - s(n2)       = n1 - n2

end natural-numbers


data module bags
begin

   parameters
      items
         begin
            sorts item
         end items

   exports
      begin
         sorts bag
         functions
           empty-bag :              -> bag
           add       : item # bag -> bag
      end

   variables
      i1, i2 : -> item
      b      : -> bag
   equations
      [1]  add(i1, add(i2, b)) = add(i2, add(i1, b))
end bags
```

```
data module set
begin

   parameters
      equality
         begin
            functions
               eq : item # item -> BOOL
         end equality

   exports
      begin
         functions
            eq      : set # set  -> BOOL
            element : item # set -> BOOL
      end

   imports
      bags
         { renamed by
               [ bag       -> set,
                 empty-bag -> empty-set]
         },
      booleans

   variables
      i, i1, i2 : -> item
      s         : -> set
   equations
      [1]  add(i, add(i, s))         = add(i, s)
      [2]  element(i, empty-set)     = false
      [3]  element(i1, add(i2, s))   = or(eq(i1, i2), element(i1, s))

end set


data module tables
begin

   parameters
      items
         begin
            sorts key, value
            functions
               eq            : key # key -> BOOL
               default-value :           -> value
         end items

   exports
      begin
         sorts table
         functions
            empty-table :                       -> table
            add         : key # value # table -> table
            look-up     : key # table           -> value
      end

   imports booleans
   variables
      k, k1, k2 : -> key
      v         : -> value
      t         : -> table
   equations
      [1]  look-up(k, empty-table)      = default-value
      [2]  look-up(k1, add(k2, v, t)) = v
                                        when eq(k1, k2) = true
      [3]  look-up(k1, add(k2, v, t)) = look-up(k1, t)
                                        when eq(k1, k2) = false

end tables
```

4.2. Data Types Specific to the Transit Node

The module *time* supplies functions to deal with timing information. To the outside the sort *time* is built up from the constant *initial-time,* using the +-function to add durations. A *duration* is either the constant *tick-duration,* or the difference of two times. Internally we use the *naturals* and auxiliary functions to define the exported functions.

```
data module time
begin

   exports
      begin
         sorts time, duration
         functions
           initial-time  :                       -> time
           tick-duration :                       -> duration
           lt            : duration # duration -> BOOL
           _ + _         : time # duration     -> time
           _ - _         : time # time         -> duration
      end

   imports natural-numbers

   functions
      time     : nat -> time
      duration : nat -> duration

   variables
      n1, n2 : -> nat
   equations
      [1]  initial-time                    = time(0)
      [2]  tick-duration                   = duration(s(0))
      [3]  lt(duration(n1), duration(n2)) = lt(n1, n2)
      [4]  time(n1) + duration(n2)         = time(n1 + n2)
      [5]  time(n1) - time(n2)             = duration(n1 - n2)

end time
```

The type of information that can be transmitted through the transit node is defined in the module *datum*.

```
data module datum
begin

   exports
      begin
         sorts datum
      end

   imports natural-numbers

   functions
      datum : nat -> datum

end datum
```

The transit nodes contains a number of ports for input and output. These ports are named with natural numbers. Port names can be collected into sets by binding the parameter of the basic module *set* to *port-name*.

```
data module port-name
begin

   exports
      begin
         sorts
            port-name
         functions
            eq : port-name # port-name -> BOOL
      end

   imports natural-numbers
      functions
         port-name : nat -> port-name

   variables
      n1, n2 : -> nat
   equations
      [1]  eq(port-name(n1), port-name(n2)) = eq(n1, n2)

end port-name


data module port-sets
begin

   imports
      set
         { renamed by
               [ set       -> port-set,
                 empty-set -> empty-port-set ]
           items bound by
               [ item      -> port-name ]
               to port-name
           equality bound by
               [ eq        -> eq ]
               to port-name
         }
end port-sets
```

A *route* consists of a *route-name* and a set of output ports associated with this route. Routes are collected into tables in order to look up the port-set corresponding to the name of a previously created route.

```
data module route-names
begin

   exports
      begin
         sorts
           route-name
         functions
           eq : route-name # route-name -> BOOL
      end

   imports natural-numbers
         functions
           route-name : nat -> route-name

   variables
      n1, n2 : -> nat
   equations
      [1]  eq(route-name(n1), route-name(n2)) = eq(n1, n2)

end route-names


data module routes
begin

   exports
      begin
         sorts route
         functions
           route    : route-name # port-set -> route
           name-of  : route                 -> route-name
           ports-of : route                 -> port-set
           eq       : route # route         -> BOOL
      end

   imports booleans, port-sets, route-names

   variables
      n1, n2   : -> route-name
      ps1, ps2 : -> port-set
   equations
      [1]  name-of(route(n1, ps1))          = n1
      [2]  ports-of(route(n1, ps1))         = ps1
      [3]  eq(route(n1, ps1), route(n2, ps2)) =
                           and(eq(n1, n2), eq(ps1, ps2))

end routes
```

```
data module route-tables
begin

   imports
     tables
        {renamed by
           [ table         -> route-table,
             empty-table   -> empty-route-table]
         items bound by
           [ key           -> route-name,
             value         -> port-set,
             eq            -> eq,
             default-value -> empty-port-set]
           to routes}

end route-tables
```

If components communicate to the outside world or to each other, messages are exchanged. Most of the messages are indexed with a value of some data type. Messages can be collected in bags.

```
data module messages
begin

   exports
     begin
       sorts message
       functions
         add-datum-port   : port-name          -> message
         add-route        : route              -> message
         send-faults      :                    -> message
         routed-datum     : route-name # datum -> message
         req-route        : route-name         -> message
         available-ports  : port-set           -> message
         timed-message    : time # datum       -> message
         datum            : datum              -> message
     end

   imports datum, time, port-name, routes

end messages


data module message-bags
begin
   imports
     bags
        { renamed by
              [ bag       -> message-bag,
                empty-bag -> empty-message-bag ]
          items bound by
              [ item      -> message ]
              to messages
        }
end message-bags
```

The various components of the transit node are connected to each other with *channels*. There are also channels to the environment.

```
data module channels
begin

   exports
      begin
         sorts channel
         functions
            control-input      :                         -> channel
            control-output     :                         -> channel
            control-in-to-out  :                         -> channel
            control-to-data    : port-name               -> channel
            data-to-control    : port-name               -> channel
            rejection          :                         -> channel
            data-in-to-out     : port-name # port-name -> channel
            data-input         : port-name               -> channel
            data-output        : port-name               -> channel
      end

   imports port-name

end channels
```

4.3. The Processes

4.3.1. Communication

The module *communication* defines the atomic actions that can be executed by the various components, when trying to communicate. The communication function is defined such that a read action (*r*) and a send action (*s*) can be combined into a communication action (*c*). These actions are indexed with the channel used to communicate and the message to be transmitted. In the same way timing information can be communicated.

The set of internal actions (I) and the set of actions to be encapsulated in order to get only successful communication (H) are also defined.

```
process module communication
begin

   exports
      begin
         atoms
            r                : channel # message
            s                : channel # message
            c                : channel # message
            read-time        : time
            send-time        : time
            comm-time        : time
            sets of channel
               internal-channels =
            { control-in-to-out, rejection,
              data-to-control(pn1), control-to-data(pn1),
              data-in-to-out(pn1, pn2) | pn1 in port-name, pn2 in port-name }
            of atoms
               I = { c(c, m), comm-time(t) |
                     t in time, c in internal-channels, m in message }
               H = { r(c, m), s(c, m), send-time(t), read-time(t) |
                     t in time, c in internal-channels, m in message }
         end

      imports
         channels,
         messages,
         time

      communications
         r(c, m) | s(c, m) = c(c, m)
            for c in channel, m in message
         read-time(t) | send-time(t) = comm-time(t)
            for t in time

   end communication
```

4.3.2. Data-ports-in

For every *port-name* a process *data-port-in* is defined. Every *data-port-in* behaves as follows. First it reads from its input channel the message to send some datum along some route. Then it reads the current time and asks the *control-port-in* for the port set attached to the requested route. Then a transit attempt is made. If the route-name was faulty, an empty-port-set was returned and the incoming message is routed to the rejection channel, thus becoming faulty. If the port-set was not empty, one port is selected randomly and after adding a time stamp the incoming message is routed to that port. The process *transit-datum* is not defined in case the port-set is empty. This means that it equals deadlock.

```
process module data-ports-in
begin

   exports
      begin
         processes
            data-port-in : port-name
      end

   imports
      port-sets,
      route-names,
      time,
      communication
      processes
         transit-attempt :
                    port-set # port-name # time # route-name # datum
         transit-datum   : port-set # port-name # time # datum

      variables
         t1, t2 : -> time
         p1, p2 : -> port-name
         rn     : -> route-name
         ps     : -> port-set
         d      : -> datum

   definitions
      data-port-in(p1) =
            sum(d in datum,
             sum(rn in route-name,
              r(data-input(p1), routed-datum(rn, d)) .
              sum(t1 in time,
               read-time(t1) . s(data-to-control(p1), req-route(rn)) .
               sum(ps in port-set,
                 r(control-to-data(p1), available-ports(ps)) .
                 transit-attempt(ps, p1, t1, rn, d) .
                 data-port-in(p1)))))

      transit-attempt(empty-port-set, p1, t1, rn, d) =
            s(rejection, routed-datum(rn, d))
      transit-attempt(add(p2, ps), p1, t1, rn, d) =
            transit-datum(add(p2, ps), p1, t1, d)

      transit-datum(add(p2, ps), p1, t1, d) =
            s(data-in-to-out(p1, p2), timed-message(t1, d)) +
            transit-datum(ps, p1, t1, d)

   end data-ports-in
```

4.3.3. Data-ports-out

The following module is parameterized with a duration, max-transit-time, that determines the maximum time a message may stay within the transit node.

For every *port-name* a process *data-port-out* is defined. Every *data-port-out* is indexed with a bag of messages that must be sent to the environment. Initially this bag is empty. It starts by reading a timed message from one of the data-input-ports. This message is added to the bag and the process starts again. If the bag is not empty, the process also has the possibility to output some message from the bag. If the max-transit-time is expired, then the message becomes faulty and will be sent to the rejection channel. Otherwise, the message is sent to the environment.

```
process module data-ports-out
begin

   parameters
      max-transit-time
         begin
            functions
               max-transit-time : -> duration
         end max-transit-time

   exports
      begin
         processes
            data-port-out : port-name # message-bag
      end

   imports
      port-name,
      message-bags,
      communication

   processes
      handle-message-out : BOOL # datum # port-name

   variables
      t, t1, t2 : -> time
      p1, p2    : -> port-name
      mb        : -> message-bag
      d, e      : -> datum

definitions
   data-port-out(p2, empty-message-bag) =
         sum(p1 in port-name,
          sum(t1 in time,
           sum(d in datum,
            r(data-in-to-out(p1, p2), timed-message(t1, d)) .
            data-port-out(p2, add(timed-message(t1, d),
                           empty-message-bag)))))
   data-port-out(p2, add(timed-message(t2, e), mb)) =
         sum(p1 in port-name,
          sum(t1 in time,
           sum(d in datum,
            r(data-in-to-out(p1, p2), timed-message(t1, d)) .
            data-port-out(p2, add(timed-message(t1, d),
                           add(timed-message(t2, e), mb)))))) +
         sum(t in time,
          read-time(t) .
          handle-message-out(lt(t - t2, max-transit-time), e, p2) .
          data-port-out(p2, mb))

   handle-message-out(false, d, p2) =
         s(rejection, datum(d))
   handle-message-out(true, d, p2) =
         s(data-output(p2), datum(d))

end data-ports-out
```

4.3.4. Control-port-in

The process *control-port-in* keeps track of all defined routes and all existing ports, so it is indexed with a *route-table* and a *port-set*. It is connected to the environment with the *control-input* channel. Via this channel it can receive the message to add a datum-port, to add a route, or to flush all faulty messages. As a last option it can receive a request from some *data-port-in* to send the routing information belonging to some *route-name*. All these incoming messages are treated separately. The request to add a datum port is handled using a subprocess. This handler checks wether the data port already exists. Then it either rejects the message or adds the port to the *port-set* and creates two new parallel processes: a *data-port-in* and a *data-port-out*.

If a request is made to add a route, it simply adds the route information to the *route-set*. A *send-faults* request is simply passed on to the *control-port-out*. A request for route information is answered by looking up the requested information and sending it back.

```
process module control-port-in
begin

   exports
      begin
         processes
            control-port-in : route-table # port-set
      end

   imports
      route-tables, communication, data-ports-in, data-ports-out

   processes
      handle-add-port : route-table # port-set # port-name # BOOL

   variables
      p  : -> port-name
      rt : -> route-table
      ps : -> port-set

definitions
   control-port-in(rt, ps) =
         sum(p in port-name,
          r(control-input, add-datum-port(p)) .
          handle-add-port(rt, ps, p, element(p, ps)))
       + sum(r in route,
          r(control-input, add-route(r)) .
          control-port-in(add(name-of(r), ports-of(r), rt), ps))
       + r(control-input, send-faults) .
         s(control-in-to-out, send-faults) .
         control-port-in(rt, ps)
       + sum(p in port-name,
          sum(rn in route-name,
          r(data-to-control(p), req-route(rn)) .
          s(control-to-data(p),
             available-ports(look-up(rn, rt))))) .
           control-port-in(rt, ps)
      handle-add-port(rt, ps, p, true) =
            s(rejection, add-datum-port(p)) .
            control-port-in(rt, ps)
      handle-add-port(rt, ps, p, false) =
            control-port-in(rt, add(p, ps)) ||
               data-port-in(p) || data-port-out(p, empty-message-bag)

   end control-port-in
```

4.3.5. Control-port-out

The process control-port-out is indexed with the *message-bag* containing all faulty messages. It has a simple behaviour. It can receive the message to send all faulty messages to the environment, which is handled by the subprocess *flush*, or it can receive faulty message via the rejection channel.

```
   process module control-port-out
   begin

      exports
         begin
            processes
               control-port-out : message-bag
         end

      imports
         message-bags,
         communication

      processes
         flush : message-bag

      variables
         m  : -> message
         mb : -> message-bag

   definitions
      control-port-out(mb) =
            r(control-in-to-out, send-faults) . flush(mb)
          + sum(m in message, r(rejection, m) .
               control-port-out(add(m, mb)))

      flush(empty-message-bag) = control-port-out(empty-message-bag)
      flush(add(m, mb)) = s(control-output, m) . flush(mb)

   end control-port-out
```

4.3.6. Transit-node

Finally the transit node is specified by the concurrent operation of the *clock* process, which is a parameter of the system, the *control-port-in* and the *control-port-out*. These ports are initialized with an empty table, set and bag. In order to hide internal actions and to get only successful communication, we add the hiding operator and the encapsulation operator.

Note that apart from the parameter *clock*, we also inherit the parameter *max-transit-time* from the imported module *data-ports-out*.

```
process module transit-node
begin

   parameters
      time
         begin
            processes
               clock
         end time

   exports
      begin
         processes
            transit-node
      end

   imports
      control-port-in,
      control-port-out

definitions
   transit-node = hide(I, encaps(H,
         clock ||
         control-port-in(empty-route-table, empty-port-set) ||
         control-port-out(empty-message-bag)))

end transit-node
```

4.4. EXAMPLE OF A CLOCK

In this section we give an example of how the clock parameter of the transit node can be initialized. The process *clock* starts at the *initial-time*. Then it can do a *tick*, followed by an increment of the current time with a *tick-duration*, or it can send the time to anyone willing to read it. Note that in this version of a clock the action of sending the time will not cost any time.

```
process module  a-clock
begin

   exports
      begin
         processes
            clock
      end

   imports
      time,
      communication
   atoms
      tick

   processes
      clock : time

   variables
      t : -> time
   definitions
      clock    = clock(initial-time)
      clock(t) = tick . clock(t + tick-duration) +
                 send-time(t) . clock(t)

end a-clock


process module  transit-node-with-a-clock
begin

   imports
      transit-node
          {time bound by
             [clock -> clock]
          to a-clock}

end transit-node-with-a-clock
```

4.5. Graphical Representation of the Import Relation

Using the IDEAS tool developed within the METEOR project [64] we can give the following picture, see figure 2), representing the import relation between all modules of the specification of the transit node. Rectangular boxes are used for data modules and boxes with rounded corners are used for process modules. An arrow from a module to another module means that the former is imported into the latter. Note that not all textual imports are present in the picture. We used a tool to compute the minimal import relation having the same transitive closure as the textual one.

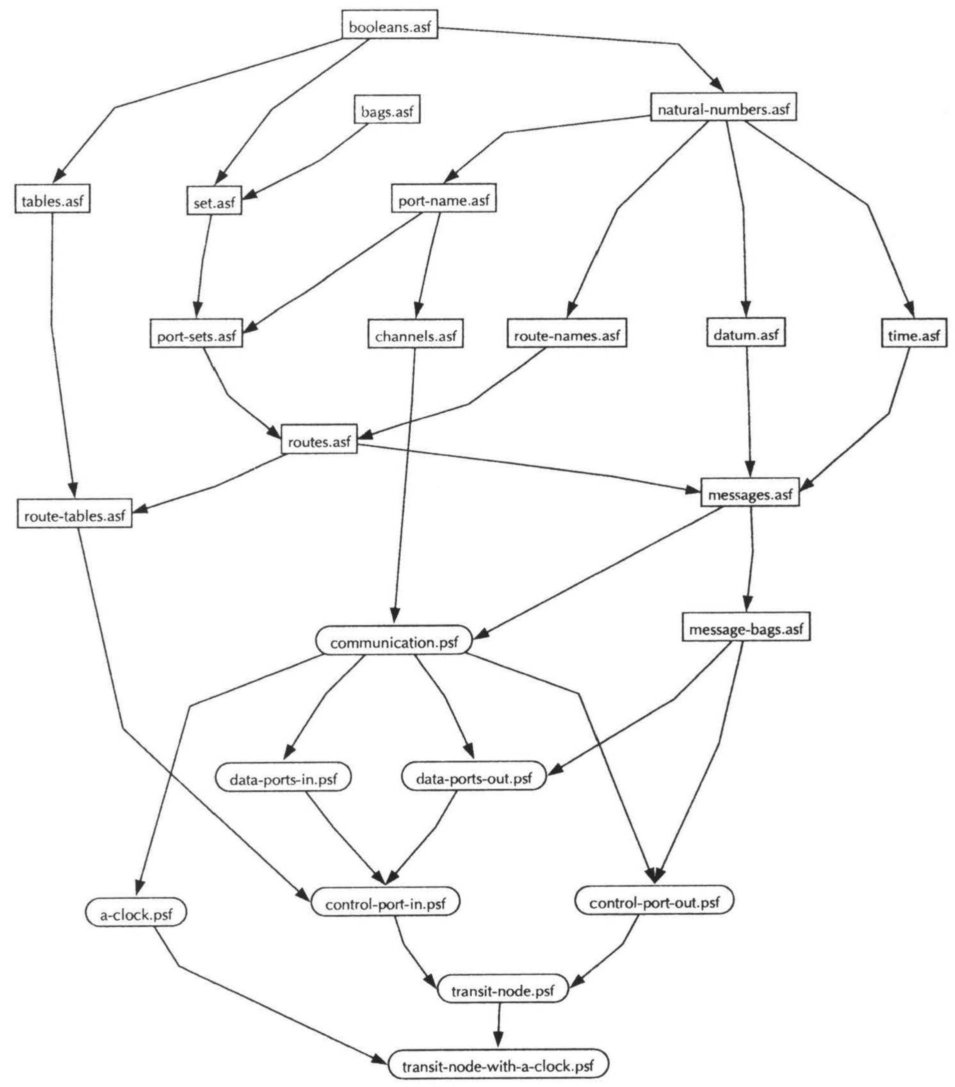


figure 2 The import relation

5. Relation to the ERAE Specification

In this section we will give a brief discussion of the relation between the ERAE specification and the PSF specification of the transit node. It is clear that, since ERAE was designed for requirements specification, the former is closer to the textual specification, whereas in the PSF specification some design decisions had to be made. As an example look at the routing information that is treated as a separate entity in ERAE, while in PSF it is part of the state of the *control-port-in*.

The ERAE language is based on temporal logic. Its formal semantics can be found in [55], and [39] contains an introduction to the use of ERAE.

In order to validate that a PSF specification is correct with respect to an ERAE specification, a formal treatment of this notion of validation would be needed. Since this chapter does not focus on this subject, we only give some informal reasoning about the relation between the two specifications.

Such a validation is made up of two parts. First we must give a relation between the entities declared in the ERAE specification and the ones declared in the PSF specification, and then we must provide an interpretation of the temporal statements in ERAE into PSF.

5.1. Entities

A quick inspection learns that, apart from some design decisions and detail implementations, the entities in ERAE relate to the entities in PSF having the same name. So where ERAE contains messages such as *Add-route msgs* indexed with a *route nr* and a series of *out port-nr*, PSF has a data type *messages*, containing a function *add-route*, indexed with *route* which is a combination of a *route-name* and a *port-set*.

As an other example look at the entity *Data port-in* which is indexed with a *nr*, and is able to receive *Data msgs* via a *port*. In PSF this translates to a process *data-port-in*, indexed with a *port-name*, having a channel to the environment called *data-input*, via which it can receive a *routed-datum*.

5.2. Temporal Statements

Naively speaking the interpretation of a temporal statement in ERAE into PSF consists of an interpretation of all events involved into atomic actions, followed by a verification that every possible trace of the specification in PSF satisfies all temporal statements about events given in the ERAE specification. Unfortunately this approach is too simple since not only temporal information is involved but also information about the state space of the system.

As an example of how to informally validate the PSF specification, we will give some ERAE statements and their informal interpretation in the PSF specification.

```
initially ⇒  ¬∃ dpi: is-in(dpi, Data-ports-in)
                    ∧     ¬∃ dpo: is-in(dpo, Data-ports-out)
                    ∧     ¬∃r: is-in(r, Routes)
                    ∧     ¬∃ wm,dm: faulty(wm) ∨ faulty(dm)
```

This can be interpreted as the statement that there are no data ports in the definition of the process *transit-node*, and that the *port-set*, *route-table* and (faulty) *message-bag* are empty:

```
transit-node = hide(I, encaps(H,
        clock ||
        control-port-in(empty-route-table, empty-port-set) ||
        control-port-out(empty-message-bag)))
```

A number of statements are about the behaviour of the environment of the transit node. These statements are not explicitly met by the PSF specification, since it only specifies the behaviour of the transit node without restricting its environment. As an example look at the statement

```
occurs(dm) ⇒ ● exists(port(dm))
```

which states that messages only arrive at existing input ports (the symbol ● means "true in the previous state"). This assumption about the environment is not stated in the PSF specification.

As a last example look at the statement about state changes concerning *data-ports-in*:

```
exists(dpi) ∧ ● ¬ exists(dpi)
  ⇒ ∃ apm: occurs(apm) ∧ nr(dpi)=port-nr(apm)
```

This states that if a *data-port-in* is created, an add-port-message must have been occurred. In the PSF specification this is verified by looking at all places where a *data-port-in* is created. This can only happen in the subprocess *handle-add-port* of the process *control-port-in*. This subprocess is only invoked after the atomic action *c(control-input, add-datum-port(p))* has occurred for some appropriate *port-name p*.

It is clear that this reasoning is very informal. This is because the existence of a data-port-in is easy to check at the textual level of the specification, but not at the level of the semantics of PSF. The semantics is a labeled transition graph, which in no way contains information about the number of processes that it is constructed from, but only about the actions that can be performed by the system. Also the actual value of the indexes of the processes involved is not part of the semantics.

6. DISCUSSION

Since some design decisions were needed, the specification of the transit node in PSF is more specific than the specification in ERAE. There is no easy transformation from an ERAE specification to a PSF specification, however when having an ERAE specification, the informal text can be interpreted more easily.

We can only give an informal validation of the PSF specification when relating it to the ERAE specification. This is due to the fact that in some cases ERAE statements relate to the state of the system, which is not part of the formal semantics of PSF. We can however look at the textual level of the specification and give an informal reasoning. Also restrictions to the environment can not be expressed in PSF.

The design of the specification can be generalized to the following method:

- Identify the parameters of the system.
- Identify all concurrent components.
- Add indexes to the process names of each component to keep track of state information and to create more instances of the object.
- Define the abstract data types needed for these indexes.
- Specify how the components are connected.
- Define the initial state of the system.
- Define the behaviour of each component.

Of course the last step of this method can be very involved. Each component in turn can then be divided into subcomponents, in such a way that the method recursively applies to these subcomponents.

REFERENCES

[1] A.V. Aho & J.D. Ullman, *Principles of compiler design*, Addison-Wesley, Reading, Massachusetts, 1977.

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

[3] R. Azarhoosh, *A PSF specification for triangular systems of equations*, Master's thesis, Programming Research Group, University of Amsterdam, 1991.

[4] J.C.M. Baeten (ed.), *Applications of process algebra*, Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[5] J.C.M. Baeten & J.A. Bergstra, *Global renaming operators in concrete process algebra*, Information & Computation 78, pp. 205-245, 1988.

[6] J.C.M. Baeten & J.A. Bergstra, *Real time process algebra*, Formal Aspects of Computing 3 (2), pp.142-188, 1991.

[7] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Syntax and defining equations for an interrupt mechanism in process algebra*, Fundamenta Informaticae IX (2), pp. 127-168, 1986.

[8] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *Conditional axioms and α/β calculus in process algebra*, Proc. IFIP Conf. on Formal Description of Programming Concepts - III, Ebberup 1986, (M. Wirsing, ed.), North-Holland, Amsterdam, pp. 53-75, 1987.

[9] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, *On the consistency of Koomen's Fair Abstraction Rule*, Theoretical Computer Science 51 (1/2), pp.129-176, 1987.

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

[11] 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.), Springer LNCS 490, pp. 303-335, 1991.

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

[13] J.C.M. Baeten & F.W. Vaandrager, *Specification and verification of a circuit in ACP*, report P8821, Programming Research Group, University of Amsterdam, 1988.

[14] J.C.M. Baeten & W.P. Weijland, *Process algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

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

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

[17] J.A. Bergstra, *A mode transfer operator in process algebra*, report P8808, Programming Research Group, University of Amsterdam, 1988.

[18] J.A. Bergstra, *A process creation mechanism in process algebra*, in: Applications of process algebra (J.C.M. Baeten, ed.), in [4], pp. 81-88, 1990.

[19] J.A. Bergstra, J. Heering & P. Klint, *Algebraic definition of a simple programming language*, report CS-R8504, CWI, Amsterdam, 1985.

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

[21] J.A. Bergstra, J. Heering & P. Klint, *Module algebra*, Journal of the Association for Computing Machinery 37(2), pp. 335-372, 1990.

[22] J.A. Bergstra & J.W. Klop, *Process algebra for synchronous communication*, Information & Control 60 (1/3), pp. 109-137, 1984.

[23] J.A. Bergstra & J.W. Klop, *Algebra of communicating processes with abstraction*, TCS 37 (1), pp. 77-121, 1985.

[24] J.A. Bergstra & J.W. Klop, *Process algebra: specification and verification in bisimulation semantics*, in: Math. & Comp. Sci. II, (M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens, eds.), CWI Monograph 4, pp 61-94, North-Holland, Amsterdam, 1986.

[25] J.A. Bergstra & J.W. Klop, *Verification of an alternating bit protocol by means of process algebra*, in: Math. Methods of Spec. & Synthesis of Software Systems '85, (W. Bibel & K.P. Jantke, eds.), Math. Research 31, Akademie-Verlag Berlin, pp 9-23, 1986.

[26] J.A. Bergstra, J.W. Klop & E.-R. Olderog, *Readies and failures in the algebra of communicating processes*, in: SIAM J. of Comp. 17(6), pp. 1134-1177, 1988.

[27] J.A. Bergstra, J.W. Klop & J.V. Tucker, *Process algebra with asynchronous communication mechanisms*, in: Proc. Seminar on Concurrency (S.D. Brookes, A.W. Roscoe & G. Winskel, eds.), LNCS 197, Springer Verlag, pp. 76-95, 1985.

[28] J.A. Bergstra, S. Mauw & F. Wiedijk, *Uniform algebraic specifications of finite sets with equality*, Int. J. of Foundations of Computer Science 2 (1), pp. 43-65, 1991.

[29] F. Biemans, *Reference model of production control systems*, Proc. of the IECON 86, Milwaukee, 1986.

[30] F. Biemans & P. Blonk, *On the formal specification and verification of CIM architectures using LOTOS*, Computers in Industry 7(6), pp. 491-504, 1986.

[31] B.W. Boehm, *A spiral model of software development and enhancement*, IEEE Computer 21 (5), pp. 61-72, 1988.

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

[33] J.J. Brunekreef, *A formal specification of three sliding window protocols*, report P9102, Programming Research Group, University of Amsterdam, 1991.

[34] K.L. Clark & S. Gregory, *Notes on systems programming in PARLOG*, Fifth generation computer systems 1984, pp. 299-306, North Holland, 1984.

[35] R. Cleaveland, J.G. Parrow & B. Steffen, *The concurrency workbench*, in: Proc. Workshop on Automatic Verification Methods for Finite-State Systems, LNCS 407, Springer -Verlag, Berlin, pp. 24-37, 1989.

[36] Commodore business machines Inc., *Commodore 64 programmer's reference guide*, 1982.

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

[38] E.W. Dijkstra, *A discipline of programming*, Prentice Hall, 1976.

[39] E. Dubois, J. Hagelstein & A. Rifaut, *Formal requirements engineering with ERAE*, Philips Journal of Research 43, nos. 3/4, pp. 393-414, 1988.

[40] H. Ehrig & B. Mahr, *Fundamentals of algebraic specifications, Vol. I, Equations and Initial Semantics*, Springer-Verlag, 1985.

[41] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans & G.R. Renardel de Lavalette, *Formal definition of the design language COLD-K*, Technical Report METEOR/t7/PRLE/7, 1987.

[42] J.C. Fernandez, *Aldébaran, a tool set for deciding bisimulation equivalences,*, in: Proc. CONCUR '91, (J.C.M. Baeten & J.A. Bergstra, eds.), Amsterdam, 1991.

[43] P. Franchi-Zannettacci & A. Zarli, *An incremental and graphical structure-oriented editor for G-LOTOS*, FORTE '90, Third Int. Conf. on Formal Description Techniques, Madrid, 1990.

[44] *Functional Specification and Description Language (SDL)*, CCITT, Recommendation Z.100-Z.104, Geneva, 1984.

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

[46] R.J. van Glabbeek, *Notes on the methodology of CCS and CSP*, report CS-R8624, CWI, Amsterdam, 1986

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

[48] J.A. Goguen & J. Meseguer, *Initiality, induction and computability*, in: Algebraic Methods in Semantics (M. Nivat & J.C. Reynolds eds.), pp. 460-541, Cambridge University Press, 1985.

[49] G. Goos, W.A. Wulf, A. Evans & K.J. Butler (eds.), *DIANA, an intermediate language for Ada*, LNCS 161, Springer-Verlag, 1983.

[50] J.F. Groote, *Transition systems with negative premises*, report CS-R8950, CWI, Amsterdam, extended abstract in Proc. CONCUR 90, (J.C.M. Baeten & J.W. Klop, eds.), Amsterdam, LNCS 458, pp. 332-341, Springer-Verlag, 1990.

[51] J.F. Groote & A. Ponse, *The syntax and semantics of μ-CRL*, report CS-R9076, CWI, Amsterdam, 1990.

[52] J.F. Groote & A. Ponse, *Proof theory for μCRL*, report CS-R9138, CWI, Amsterdam, 1991.

[53] J.F. Groote & F.W. Vaandrager, *An efficient algorithm for branching bisimulation and stuttering equivalence*, in: Proc. 17th ICALP, (M.S. Paterson, ed.),Warwick, LNCS 443, pp. 626-638, Springer-Verlag, 1990.

[54] J. Hagelstein, *The transit node - ERAE specification*, METEOR report, Philips Research Laboratory Brussels, 1988.

[55] J. Hagelstein & A. Rifaut, *The semantics of ERAE*, Philips Research Laboratory Brussels Manuscript, Belgium, 1989.

[56] 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.

[57] P.R.H. Hendriks, *ASF system user's guide*, report CS-R8823, CWI, Amsterdam, 1988, Extended abstract in: Conference Proceedings of Computing Science in the Netherlands, CSN'88 1, pp. 83-94, SION, 1988.

[58] P.R.H. Hendriks, *Implementation of modular algebraic specificatins*, Ph.D. thesis, University of Amsterdam, 1991.

[59] M. Hennessy, *Proving systolic systems correct*, TOPLAS 8 (3), pp. 344-387, 1986.

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

[61] C.A.R. Hoare, *Communicating sequential processes*, Prentice-Hall, 1985.

[62] A. Hodges, *Alan Turing, The enigma of intelligence*, Burnett Books Limited, 1983.

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

[64] *IDEAS interface user guide*, Centre de Recherches de la C.G.E., Marcoussis 1988.

[65] INMOS Limited, *Occam® 2 reference manual*, Prentice Hall, 1988.

[66] International Organization for Standardization, *Information processing systems - Open systems interconnection - Estelle - A formal description technique based on an extended state transition model*, ISO/TC 97/SC 21 N DP9074, 1986.

[67] International Organization for Standardization, *Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour*, IS 8807, 1989.

[68] H. Jacobsson, *Proposal for TIL*, Master's thesis, Programming Research Group, University of Amsterdam, 1989.

[69] H. Jacobsson & S. Mauw, *A Token ring network in PSF_d*, report P8914, Programming Research Group, University of Amsterdam, 1989.

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

[71] J.W. Klop, *Term rewriting systems*, to appear in: Handbook of Logic in Computer Science, Vol. 1 (S. Abramsky, D. Gabbay and T. Maibaum, eds.), Oxford University Press, 1992.

[72] H. Kodate, K. Fujii & K. Yamanoi, *Representation of FMS with petrinet graph and its application to simulation of system operation*, Robotics and Computer-Integrated Manufacturing 3(3), pp. 275-283, 1987.

[73] N. Komoda, K. Kera & T. Kubo, *An autonomous, decentralized control system for factory automation*, IEEE Trans. Comput. 17(12), pp. 73-83, 1984.

[74] K.T. Kung, *Let's design algorithms for VLSI systems*, in: Proc. Conf. VLSI: Architecture, Design, Fabrication, California Institute of Technology, 1979.

[75] K.G. Larsen, J.C. Godskesen & M. Zeeberg, *TAV, tools for automatic verification, user manual*, technical report R 90-19, Department of Mathematics and Computer Science, Aalborg University, 1989.

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

[77] H. Lin, *Pam: a process algebra manipulator*, Proc. Third Workshop on Computer Aided Verification (K.G. Larsen & A. Skou, eds.), Aalborg, 1991.

[78] E. Madelaine, J.C. Fernandez & R. De Simone, *FC: A common format representation for automata (version 2)*, deliverable D3.2.3, ESPRIT Basic Research Action 3006 CONCUR, 1991.

[79] J. Malhotra, S.A. Smolka, A. Giacalone & R. Shapiro, *Winston, a tool for hierarchical design and simulation of concurrent systems*, in: Proc. Workshop on Specification and Verification of Concurrent Systems, University of Stirling, Scotland, 1988.

[80] A. Mauboussin, H. Perdrix, M. Bidoit, M.-C. Gaudel & J. Hagelstein, *From an ERAE requirements specification to a PLUSS algebraic specification: A case study*, in: Algebraic Methods II: Theory, Tools and Applications (J.A. Bergstra & L.M.G. Feijs, eds.), Springer LNCS 490, pp. 395-431, 1991.

[81] S. Mauw, *A constructive version of the approximation induction principle*, in: Proc. SION Conf. CSN 87, Amsterdam, pp.235-252, 1987.

[82] S. Mauw, *An algebraic specification of process algebra, including two examples*, in: Algebraic Methods: Theory, Tools & Applications, Workshop Passau 1987 (M. Wirsing & J.A. Bergstra, eds.), Springer LNCS 394, pp. 507-554, 1989.

[83] S. Mauw, *Process algebra as a tool for the specification and verification of CIM-architectures*, in [4], pp. 53-80, 1990.

[84] S. Mauw & Gy. Max, *A formal specification of the Ethernet protocol*, report P9007, Programming Research Group, University of Amsterdam, 1990.

[85] 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.

[86] S. Mauw & G.J. Veltink, *A tool interface language for PSF*, report P8912, Programming Research Group, University of Amsterdam, 1989.

[87] S. Mauw & G.J. Veltink, *A process specification formalism*, Fundamenta Informaticae XIII, pp. 85-139, 1990.

[88] S. Mauw & G.J. Veltink, *A proof assistant for PSF*, Proc. Third Workshop on Computer Aided Verification (K.G. Larsen & A. Skou, eds.), Aalborg, 1991.

[89] S. Mauw & F. Wiedijk, *Specification of the transit node in PSF_d*, in: Algebraic Methods II: Theory, Tools and Applications (J.A. Bergstra & L.M.G. Feijs, eds.), Springer LNCS 490, pp. 341-361, 1991.

[90] R. Milner, *A calculus of communicating systems*, Springer LNCS 92, 1980.

[91] J.C. Mulder, *Case studies in process specification and verification*, Ph.D. thesis, University of Amsterdam, 1990.

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

[93] H. A. Oldenburger, *Tabular viewer for PSF*, Master's Thesis, University of Amsterdam, 1991.

[94] D.M.R. Park, *Concurrency and automata on infinite sequences*, in: Proc. 5th GI Conf. (P. Deussen, ed.), Springer LNCS 104, pp. 167-183, 1981.

[95] 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.

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

[97] I. Sommerville, *Software engineering*, Academic Service/Addison Wesley, 1989.

[98] SPECS, *Definition of MR, Version 1*, D.WP5.2, The SPECS Consortium, 1989.

[99] K. Ueda, *Guarded Horn Clauses*, Technical report TR-103, ICOT, Tokyo, 1985.

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

[101] F.W. Vaandrager, *Process algebra semantics of POOL*, in [4], pp. 173-236, 1990.

[102] *VDM specification language proto-standard, draft*, BSI IST/5/50, Document N-40, 1988.

[103] G.J. Veltink, *From PSF to TIL*, report P9009, Programming Research Group, University of Amsterdam, 1990.

[104] G.J. Veltink, *The PSF toolkit*, report P9107, Programming Research Group, University of Amsterdam, 1991, to appear in: Computer Networks and ISDN Systems, special issue on Tools for FDT's.

[105] G.J. Veltink, *XP, an experiment in modular specification*, to appear in: FORTE '91, Fourth Int. Conf. on Formal Description Techniques, Sydney, 1991.

[106] C. Verhoef, *On the register operator*, report P9003, Programming Research Group, University of Amsterdam, 1990.

[107] J.L.M. Vrancken, *The implementation of process algebra specifications in POOL-T*, Ph.D. thesis, University of Amsterdam, 1991.

[108] W.P. Weijland, *Synchrony and asynchrony in process algebra*, Ph.D. Thesis, University of Amsterdam, 1989.

[109] W.P. Weijland, *Verification of a systolic algorithm in process algebra*, Cambridge Tracts in Theoretical Computer Science 10, pp. 139-158, 1990.

[110] J. Zuidweg, *Concurrent system verification with process algebra*, Ph.D. thesis, University of Leiden, 1990.

Nederlandse Samenvatting

PSF - Een Proces Specifikatie Formalisme

Deze samenvatting is bedoeld voor de lezer die niet geïnteresseerd is in de technische details, maar wel benieuwd naar de sleutelbegrippen die een rol spelen in dit proefschrift, te weten *parallellisme* en *specificeren*.

1. Parallellisme

Vanaf het prille begin van de studie van computers en hun programmering werd de oplossing van een probleem gepresenteerd als een reeks van handelingen, die, één voor één uitgevoerd, tot een oplossing van het totale probleem leiden. Later zag men in dat grotere problemen beter konden worden opgedeeld in kleinere deelproblemen. Deze deelproblemen moesten echter in een strikt vastgelegde volgorde opgelost worden. Deze naïeve methode van probleemoplossen leidt meestal tot een werkend programma en sluit goed aan bij het menselijk onvermogen om grote aantallen aspekten tegelijk te overzien.

Het idee dat er ook opsplitsingen van een probleem in deelproblemen bestaan die niet per sé strikt geordend in de tijd moeten worden opgelost, heeft pas in de laatste jaren post gevat. Dit ten onrechte grotendeels door het beschikbaar komen van parallelle computerarchitekturen. De relatie tussen de onderscheiden deelproblemen is dan niet meer een relatie in tijdsvolgorde, maar bijvoorbeeld in gegevensafhankelijkheid. Deelproblemen die niet in de tijd gerelateerd zijn kunnen dan gelijktijdig (parallel) worden opgelost.

Zoals zo vaak in het vakgebied van de konstruktie van komputerprogramma's, dat *Software Engineering* wordt genoemd, wordt gegrepen naar een metafoor om de situatie te verduidelijken. Het parallel oplossen van problemen heeft bijvoorbeeld in de bouw een natuurlijk analogon. Aan een projekt zijn vaak grote groepen bouwvakkers bezig, die allen hun eigen taakstelling hebben en dus parallel hun taak kunnen uitvoeren. Alleen de inherente tijdsafhankelijkheden, zoals de eis dat de fundering eerder dan de bovenliggende verdiepingen moet zijn gerealiseerd, zorgen voor een gedeeltelijk sequentiële ordening. Ook in de bouw is het streven deze tijdsordening te minimaliseren, bijvoorbeeld door het gebruik van geprefabriceerde komponenten. Hierbij worden delen van een woning al in de fabriek gekonstrueerd. In tegenstelling tot de hardware fabricage (chips e.d.) lijkt de fabricage van software minder eenvoudig af te stemmen op het gebruik (en hergebruik) van vooraf gebouwde komponenten.

1.1. Waarom Parallel Rekenen

Er kunnen twee redenen worden onderscheiden waarom het parallel oplossen van een probleem in de informatica een rol kan spelen. De eerste is dat parallel redeneren aansluit bij de aard van het probleem en de tweede is dat parallellisme tijdwinst oplevert bij het oplossen van het probleem.

In het eerste geval is uit de probleemstelling af te leiden dat er een aantal tijdsonafhankelijke deelproblemen zijn, die geen expliciete tijdsrelatie tot elkaar hebben. Het ligt dan voor de hand om ook geen tijdsvolgorde aan de oplossing van de deelproblemen op te leggen. Het herkennen van parallellisme hangt natuurlijk sterk van de cultuur van de probleemoplosser af. Als iemand gewend is alle problemen in termen van "control-flow" te zien, zal niet direkt duidelijk zijn dat er ook een parallelle of een "object-oriented" oplossing te formuleren is. Een voorbeeld van makkelijk te herkennen parallellisme is een probleem waarbij steeds gelijkvormige bewerkingen moeten worden uitgevoerd, zoals het kwadrateren van een lange lijst getallen. Het enige argument dat er bij het oplossen van dit probleem een sequentieel algoritme gebruikt zou moeten worden is het feit dat de invoer in een lijst geordend is. Deze ordening is echter niet essentieel voor het gegeven probleem. Een ander voorbeeld waarbij de beschrijving van het probleem al een opdeling in parallelle objekten geeft is een kommunikatieprotokol. Bij kommunikatie wordt al een fysiek onderscheid gemaakt tussen de diverse komponenten, zoals de zender, de ontvanger en het kommunikatiemedium. In het geval dat de diverse komponenten zich niet in elkaars direkte nabijheid bevinden, wordt er ook wel gesproken van *gedistribueerde* gegevensverwerking.

Een tweede reden om parallellisme te willen gebruiken is snelheidswinst. Als je een aantal taken parallel uitvoert kost het in totaal minder tijd dan bij sequentieel oplossen. Sommige "real-time" toepassingen zijn alleen mogelijk dankzij het gebruik van parallelle systemen, zoals bijvoorbeeld het detecteren van subatomaire deeltjes in de huidige generatie versnellers. Zonder parallelle faciliteiten is de grote hoeveelheid meetgegevens niet te verwerken. Een voorbeeld waarbij versnelling van de gegevensverwerking leidt tot een kwalitatief beter resultaat is de weersvoorspelling. Vergroting van het aantal verwerkte meetwaarden en dus van de nauwkeurigheid van de voorspelling kan alleen bij een vergroting van de verwerkingskapaciteit. Recente wiskundige inzichten echter voorspellen dat zelfs bij een onbeperkte toename van de rekenkapaciteit de nauwkeurigheid van de weersvoorspelling nauwelijks toe zal nemen.

1.2. PROBLEMEN MET PARALLELLISME

Introduktie van parallellisme kan dan een toepassing lijken te vinden in bepaalde probleemgebieden, het introduceert echter ook een nieuwe klasse van problemen. In de praktijk is het maar zelden het geval dat een verdubbeling van het aantal computers leidt tot een halvering van de rekentijd. Het rendement hangt zeer sterk af van het soort taak dat uitgevoerd moet worden. Bij een taak die op perfecte manier op te delen is in subtaken zal het toevoegen van extra computers een hoog rendement opleveren. Denk bijvoorbeeld aan de lijst getallen die gekwadrateerd moet worden. Als je hier een extra komputer aan toevoegt hoef je er slechts voor te zorgen dat de lijst in een aantal ongeveer even grote delen wordt opgesplitst. Bij een ondeelbare taak zal toevoegen van meerdere computers juist geen enkele snelheidswinst opleveren. Hierbij wordt in de literatuur meestal de metafoor van een zwangere vrouw gebruikt. De draagtijd zal negen maanden zijn, onafhankelijk van het feit of er meerdere personen zijn die bereid zijn in de zwangerschap te delen. In sommige toepassingen zal het gebruik van meerdere komputers zelfs een snelheidsverlies tot gevolg hebben. Dit wordt veroorzaakt door de burokratie die nodig is om taken te verdelen en koördineren.

In de praktijk laat een probleem zich vaak opsplitsen in deelproblemen, maar moeten de oplossers van de deelproblemen toch in enige mate samenwerken om tot het juiste antwoord te komen. De mate van samenwerking kan worden gemeten aan de hand van de hoeveelheid kommunikatie die tussen de komponenten gevoerd wordt. Bij veel problemen weegt de tijdwinst die verkregen wordt door het werk over meerdere komponenten te verdelen op tegen deze extra kommunikatie-overlast. Soms echter levert de extra kommunikatie zoveel tijdverlies op dat het resultaat van het parallel rekenen juist langzamer tot stand komt dan bij de klassieke, sequentiële manier. In het vakgebied van de *software engineering*, dat de konstruktie van programmatuur bestudeert, wordt een vergelijkbaar probleem kernachtig aangeduid door de stelling *"adding more manpower to a late software project, makes the project later"*. Doel is dus tot een zodanige opsplitsing in deelproblemen te komen dat er een minimum aan kommunikatie nodig is.

Een ander fenomeen dat zich voordoet bij het parallel oplossen van problemen is het verschijnsel "deadlock". Een deadlock is een situatie waarbij een komponent zit te wachten op informatie van een andere komponent, terwijl die ander zelf weer zit te wachten op informatie van de eerste komponent. Deze situatie zal bij menselijke kommunikatie niet vaak voorkomen, maar bij komputers maar al te vaak. Een deadlock is een gevolg van het feit dat de kommunikatie tussen de komponenten niet korrekt is uitgedacht.

Een dergelijke deadlock-situatie zal zich niet elke keer dat het systeem werkt openbaren. Het zou zich pas na een heleboel suksesvolle bedrijfsjaren kunnen voordoen, als toevallig dàn beide komponenten op (vrijwel) hetzelfde moment informatie van elkaar willen hebben. Dit onvoorspelbare gedrag is kenmerkend voor parallel werkende systemen. Als een parallel programma meerdere keren wordt gestart met invoergegevens die precies identiek zijn, kunnen er toch verschillende resultaten uitkomen. Dit wordt veroorzaakt doordat een komponent soms een fraktie sneller is en soms een fraktie langzamer dan een andere komponent, waardoor de interaktie net iets anders kan verlopen. Dit verschijnsel, dat het resultaat niet alleen bepaald wordt door het programma en zijn invoer, maar ook door toevallige omstandigheden, noemen we "non-determinisme".

Beide verschijnselen, kommunikatie met deadlock-mogelijkheid en non-determinisme zorgen, ervoor dat het niet eenvoudig is om in te zien of een parallel programma ook korrekt zal funktioneren. Aangezien het in veel gevallen noodzakelijk is om dit te weten, zal in een theorie over parallel programmeren in ieder geval aan deze aspekten aandacht moeten worden besteed. De wiskundige theorie die de basis vormt voor dit proefschrift heet "procesalgebra".

2. SPECIFICEREN

2.1. SPECIFICEREN VAN PARALLELLE SYSTEMEN

Is er eenmaal een opsplitsing van een probleem in deelproblemen uitgedacht dan doet zich nog de vraag voor hoe het op te schrijven. De meest voor de hand liggende manier om parallelle programma's te beschrijven is gebruik te maken van een parallelle programmeertaal. Het probleem echter is dat de weinige bestaande programmeertalen met voorzieningen voor parallellisme zich niet goed lenen voor het analyseren van de erin beschreven programma's.

Een dergelijke analyse is vaak wel mogelijk in procesalgebra. Met zuiver wiskundige methoden is het mogelijk om te verifiëren of een in procesalgebra beschreven programma aan de vooraf opgestelde wensen voldoet. Procesalgebra is echter, door de wiskundige notatie en door de mogelijkheid om informeel over sommige details heen te stappen, niet geschikt om per komputer te worden verwerkt. De wens om toch procesalgebra te kunnen bedrijven leidt ertoe dat er een taal wordt ontworpen die naadloos aansluit bij de terminologie uit de procesalgebra, maar deze bezwaren niet kent. De in dit proefschrift voor dit doel ontworpen taal wordt PSF genoemd, hetgeen staat voor "Process Specification Formalism".

2.2. PROCESSEN EN GEGEVENS

PSF leent zich zowel voor het beschrijven van parallelle systemen, als voor het analyseren daarvan, waarbij eventueel van de komputer gebruik kan worden gemaakt.

Bij het beschrijven van een parallel systeem worden twee zaken onderscheiden. In de eerste plaats dient aangegeven te worden wat de struktuur van het systeem is, hoe de onderlinge verbindingen lopen en hoe de diverse komponenten met elkaar kommuniceren. Dit laatste komt tot uiting in de beschrijving van het gedrag van de afzonderlijke komponenten, ook wel processen genoemd. Het specificeren van dit gedrag vindt plaats in de zogenaamde "proces-modulen" van PSF.

In de tweede plaats moeten de gegevens beschreven worden die door de diverse komponenten verwerkt worden. De struktuur van deze gegevens en de verschillende bewerkingen op die gegevens worden gedefinieerd in de zogenaamde "data-modulen" van PSF.

Een specifikatie in PSF beschrijft dus welke processen er een rol spelen en met welke gegevens ze omgaan.

2.3. MODULEN

Behalve voor het maken van onderscheid tussen processen en gegevens dient de opdeling van een specifikatie in een aantal modulen nog een ander doel. Door middel van het opdelen van een groot probleem in kleinere problemen is het mogelijk om struktuur in het probleem aan te brengen. Deze struktuur zal bij een goede specifikatie tot gevolg hebben dat er een hiërarchie van modulen ontstaat, die op een in de specifikatie vastgelegde, wijze samenhangen. Deze techniek maakt het mogelijk om komplexe problemen op te splitsen in eenvoudiger deelproblemen die onafhankelijk van elkaar kunnen worden bestudeerd en opgelost. Bovendien wordt het hierdoor mogelijk om reeds bekende deelproblemen af te splitsen en de eerder hiervoor gevonden oplossingen opnieuw te gebruiken.

3. ANALYSEREN

Vaak is men niet alleen geïnteresseerd in een beschrijving van een systeem, maar wil men ook het inzicht hebben dat het gespecificeerde systeem voldoet aan de vooraf opgestelde wensen. Men wil bijvoorbeeld voorkomen dat er ooit een deadlock zal optreden.

Een vermoeden dat het systeem korrekt zal funktioneren verkrijgt men door het te testen. Dit testen van een specifikatie wordt simuleren genoemd. Met behulp van de komputer worden een aantal mogelijke executiepaden bekeken. Aan de hand van de aldus verkregen gegevens kan men -met een geringe mate van waarschijnlijkheid- konkluderen dat het systeem goed in elkaar zit.

Meer zekerheid biedt een verifikatie van het systeem. Dit houdt in dat er een bewijs wordt gegeven dat de specifikatie in alle omstandigheden het juiste gedrag vertoont. Bij het verifiëren wordt het gespecificeerde gedrag vergeleken met het gewenste gedrag. Als beiden, na een reeks van wiskundige manipulaties, identiek blijken, is het gespecificeerde systeem korrekt. Ook hier kan dankbaar gebruik worden gemaakt van de komputer.

4. Dit Proefschrift

In dit proefschrift worden bovenstaande aspekten op een grondiger manier beschreven. In het eerste hoofdstuk wordt kort ingegaan op de redenen voor het ontstaan van PSF. Vervolgens wordt in hoofdstuk 2 de specifikatietaal PSF beschreven, waarna in hoofdstuk 3 mogelijke uitbreidingen van deze taal aan de orde komen.

In hoofdstuk 4 wordt de taal TIL (Tool Interface Language) beschreven. Deze zogenaamde tussen-taal maakt het eenvoudiger om komputerprogramma's te schrijven die PSF-specifikaties analyseren. Met behulp van TIL wisselen de verschillende programma's informatie uit, in het bijzonder de specifikatie zelf.

Tot slot wordt een tweetal voorbeelden gegeven waarin het gebruik van PSF wordt toegelicht. In het eerste voorbeeld (hoofdstuk 5) wordt een bewijs gepresenteerd dat een gegeven specifikatie van een fabrieksarchitektuur korrekt is. In het tweede voorbeeld (hoofdstuk 6) wordt geschetst hoe men uitgaande van een beschrijving van het probleem van de transit-node tot een PSF-specifikatie kan komen.