A Process Specification Formalism based on static COLD

J.C.M. Baeten[1] J.A. Bergstra[2, 3] S. Mauw[2] G.J. Veltink[2]

1: Department of Software Technology, Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam

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

3: Department of Philosophy, State University of Utrecht
Heidelberglaan 2, 3584 CS Utrecht

Abstract:

PSF/C is a formal specification language, based on COLD, a wide spectrum specification language developed at Philips Research, Eindhoven. In PSF/C, we can specify concurrent communicating processes. The process syntax and semantics is based on the algebraic concurrency language ACP.

Note: The first three authors are sponsored by ESPRIT contract 432, A Formal Integrated Approach to Industrial Software Development (METEOR). The first and second author are also sponsored by RACE contract 1046, Specification and Programming Environment for Communication Software (SPECS).

ACKNOWLEDGEMENT

The writers would like to thank Jan Rekers for assistance concerning the use of the SDF system.

1 INTRODUCTION

PSF/C is an *experiment* in language design. It is not meant as a finished language that would justify the substantial efforts of writing its necessary tools. PSF/C is a language in which we can specify concurrent communicating processes. Moreover, we have ample facilities to specify data types. These data types can occur as parameters of actions and processes. Also, we have a modular structure: data types and processes are defined in modules. Parts of the signature of these modules can be exported or hidden. The starting point for construction of PSF/C has been the wide spectrum language COLD, developed at Philips Research, Eindhoven. From COLD, we get data type specifications and the modular structure with imports and exports. On top of that, we specify processes and their interaction in the spirit of the concurrency theory ACP of [BK84].
The design objectives have been:

- to combine ACP and the *static part* of COLD in one language where the concrete syntax is borrowed from COLD;
- to combine processes and data in a similar fashion as is done in PSF/ASF of [MV88], where data are used as parameters of actions and process names;
- to obtain a semantic description of the language by means of a translation to COLD;
- to generate a parser for the syntax by means of the SDF system of the Esprit project 2177 (GIPE). (see [BHK89]).

2 THE COLD-S LANGUAGE

In this section we will present COLD-S, which is obtained by dropping all dynamic features from the language COLD-K and leaving out renamings and parameterization. In fact this language is a subset of COLD-K^2 in RENARDEL DE LAVALETTE [RdL89]. The language COLD-K has been developed in the framework of ESPRIT project 432, METEOR (see FEIJS, JONKERS, KOYMANS & RENARDEL DE LAVALETTE [FJKR87] or [WB89]). COLD-K has been designed to be a so-called wide spectrum language in which it should be possible to capture the whole spectrum of software development. The language supports *transformational* design, in which implementations are constructed from specifications by replacing, step by step, all parts of the specification by equivalents that show more and more aspects of an executable language.
Like COLD-K, COLD-S is defined by means of a translation of its grammatical constructs to the constructs of a layered formal language. Because COLD-S does not have the parameterization concept of COLD-K, the top layer of this kernel, $\lambda\pi$, is left out and so COLD-S only uses two instead of three layers. Expressions in COLD-S start off with terms from a special many-sorted algebra, called CA for Class Algebra, which is used for modeling 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$_\omega$. Every construct in a COLD specification corresponds with an expression in the kernel of formal languages with a well-defined semantics. COLD specifications are translated by means of attribute grammars to the kernel.
In some cases, we want to restrict COLD-K in another way, by taking the *algebraic* subset COLD-K^2 as described in [RdL89]. We obtain COLD-K^2 by restricting all axioms in the language to the format of *conditional equations*, and restricting all functions to *total* functions. Obviously, COLD-SA will be the static algebraic part of COLD-K.

2.1 Some Remarks on the Language

Like COLD-K, the language COLD-S consists of a number of hierarchically ordered sublanguages. This hierarchy is illustrated by the following picture:

$$\begin{tikzcd}
\text{Design Language} \arrow[d] \\
\text{Scheme Language} \arrow[d] \\
\text{Class Language} \arrow[d] \\
\text{Definition Language} \arrow[d] \\
\text{Assertion Language}
\end{tikzcd}$$

In the following sections we will explain each language in some more detail.

2.1.1 The Assertion Language

In the assertion language we can write terms and assertions. The assertions in COLD-K or COLD-S are exactly the formulae of MPL, the underlying many-sorted predicate logic. In the case of COLD-K^2 we only allow (universally quantified) conditional equations.

2.1.2 The Definition Language

In the definition language we come across the items that are defined in the COLD-S language, viz.: sorts, predicates and functions. A definition can be seen in two ways: a declarative and a definitional way. The declarative part introduces the name of an item and possibly its type, while the definitional part defines the meaning of the item introduced. Not all definitions show both aspects. Sort definitions only have a declarative aspect, while axioms are purely definitional. Predicates and functions are both declarative and definitional, their meaning is defined directly, by a defining term or an assertion, or indirectly, by an inductive definition or an axiom. Inductively defined predicates and functions are defined as the smallest predicate or function satisfying the inductive definition.

2.1.3 The Class Language

The class language is used to group a list of definitions into a modular structure which is called *class* in COLD-S. The *signature* of a class is the collection of sorts, functions and predicates that are defined in that particular class.

2.1.4 The Scheme Language

All operations that have to do with the modularization of specifications are dealt with in the scheme language.

These operations are :

- import of classes
- export of objects from a class
- introduction of abbreviations

2.1.5 The Design Language

The design language is used to handle specifications at the highest level. At this level the so-called components, which will finally be used to specify the complete *system*, are specified. A component can be either a specification, in which case it is called a *specified* component, or a specification together with an implementation written in COLD-S, in which case it is called an *implemented* component. Specified components are used when the implementation of a component cannot be described in COLD-S, because it is a piece of hardware or an existing program in some kind of programming language.

2.2 The Grammar

The definition of the context free grammar of COLD-S is given using a certain BNF-grammar augmented with the following extra rules:

{X}	denotes zero or more occurrences of X (a list of X's)
[X]	denotes zero or one occurrences of X (an optional X)
{ X '@' }	denotes zero or more occurrences of X. The symbol @ acts as delimiter.

Then, the grammar of COLD-S is defined as follows:

```
<design> ::= DESIGN {<component> ';'} SYSTEM {<scheme> ','}

<component> ::= COMP <scheme-var> : <scheme> [:= <scheme>]
   | LET <scheme-var> := <scheme>

<scheme> ::= <class>
   | IMPORT <scheme> INTO <scheme>
   | EXPORT <signature> FROM <scheme>
   | LET <scheme-var> := <scheme> ; <scheme>
   | <scheme-var>

<signature> ::= {<item> ','}
   | <signature> + <signature>
   | <item> ^ <signature>
   | SIG <scheme>
```

```
<item> ::= SORT <sort-name>
   | PRED <predicate-name> : domain
   | FUNC <function-name> : domain -> <sort-name>

<class> ::= CLASS {<definition>} END

<definition> ::= SORT <sortname>
   | PRED <predicate-name> : domain <predicate body>
   | FUNC <function-name> : domain -> <sort-name> <function body>
   | AXIOM <assertion>

<predicate body> ::= [IND <assertion>]
   | [PAR <varsort list>] DEF <assertion>

<function body> ::= [IND <assertion>]
   | [PAR <varsort list>] DEF <term>

<assertion> ::= TRUE
   | FALSE
   | <term>!
   | <term> = <term>
   | <predicate-name> <term list>
   | NOT <assertion>
   | <assertion> ; <assertion>
   | <assertion> AND <assertion>
   | <assertion> OR <assertion>
   | <assertion> => <assertion>
   | <assertion> <=> <assertion>
   | FORALL <varsort list> <assertion>
   | EXISTS <varsort list> <assertion>
   | LET {<assignment> ','} ; <assertion>
   | ( <assertion> )

<term> ::= <object-var>
   | <function-name> <term list>
   | THAT <varsort> <assertion>
   | LET {<assignment> ','} ; <term>
   | ( <term> )

<term list> ::= {<term> ','}
   | ( <term list> )

<domain> ::=  {<sort-name> '#'}

<varsort list> ::= {<varsort> ','}

<varsort> ::= <object-var> : <sort-name>

<assignment> ::= <object-var> := <term>

<scheme-var> ::= <identifier>

<sort-name> ::= <identifier>

<predicate-name> ::= <identifier>

<function-name> ::= <identifier>

<object-var> ::= <identifier>
```

3 PSF/C

The concrete syntax of PSF/C is almost identical to the concrete syntax of COLD, with the exception of the additional language constructs we need to represent atomic actions, processes etc. To indicate we restrict ourselves to the static part of COLD, COLD-S, we write PSF/C. Similarly, for PSF/CA we use the static algebraic part of COLD, COLD-SA.

3.1 Character Set

A PSF/C specification uses the same ASCII character set as COLD, viz. :

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 :
; < = > ? @ A B C D E F G H I J K L M N O P Q R S T U
V W X  Y Z [ \ ] ^ _ ` a b c d e f g h i j k l m n o p
q r s t u v w x y z { | } ~
```

3.2 Tokens

In parsing a PSF/C specification a series of tokens is recognized. Each token is a sequence of ASCII characters and tokens are separated by spaces, tabs and new lines. In cases of ambiguity the longest token that can be recognized is preferred. There are three kinds of tokens, viz. identifiers, keywords and comments. We will discuss these in turn in the following sections.

3.2.1 Identifiers

Identifiers in PSF/C are arbitrary non-empty strings consisting of letters, digits and the following four characters:

" ' / _

excluding those strings which are keywords. Two characters that can be part of a COLD identifier are excluded namely the dot '.' and the backslash '\'. The dot has become a keyword, representing sequential composition and the backslash is reserved to be used as a special character that a program translating PSF/C into COLD-K can use to distinguish user defined identifiers from identifiers generated by the translator.

3.2.2 Keywords

The following strings are PSF/C keywords:

!	^	FORALL	PRETAU
#	ACTION	FROM	PROCESS
&	AND	FUNC	SET
(	AXIOM	GCMD	SIG
)	CLASS	HIDE	SORT
+	COMM	IMPORT	SPEC
,	COMP	IND	SUM
->	DEF	INTO	SYSTEM
.	DELTA	LET	THAT
:	DESIGN	MERGE	TRUE
:=	ENCAPS	NOT	WITH
;	END	OF	|
<=>	EXISTS	OR	||
=	EXPORT	PAR	
=>	FALSE	PRED	

3.2.3 Comments

There are two possible ways to create a comment. The first is to use the comment brackets: '{' and '}', which turn the enclosed text into a comment. Comment brackets cannot be nested and the enclosed text may not contain a '}'.
Example:

{ This is a comment }

The second way to create comment is by using the sign the '%', which turns the rest of the line into a comment.
Example:

% This is comment

Comments may be inserted between any two tokens and have no meaning in terms of the abstract syntax.

3.3 Grammar

The PSF/C grammar is given in the following section. In fact it is an extension of the COLD-S grammar presented in section 2.

```
<design> ::= DESIGN {<component> ';'} SYSTEM {<scheme> ','}

<component> ::= COMP <scheme-var> : <scheme> [:= <scheme>]
   | LET <scheme-var> := <scheme>

<scheme> ::= <class>
   | IMPORT <scheme> INTO <scheme>
   | EXPORT <signature> FROM <scheme>
   | LET <scheme-var> := <scheme> ; <scheme>
   | <scheme-var>

<signature> ::= {<item> ','}
   | <signature> + <signature>
   | <item> ^ <signature>
   | SIG <scheme>

<item> ::= SORT <sort-name>
   | PRED <predicate-name> : domain
   | FUNC <function-name> : domain -> <sort-name>
   | ACTION <action-name>  : domain
   | PROCESS <process-name>  : domain
   | SET <set-name>

<class> ::= CLASS {<definition>} END

<definition> ::= SORT <sortname>
   | PRED <predicate-name> : domain <predicate body>
   | FUNC <function-name> : domain -> <sort-name> <function body>
   | AXIOM <assertion>
   | ACTION <action-name> : domain
   | PROCESS <process-name> : domain  <process body>
   | SET <set-name> <set body>
   | COMM <comm assertion>
   | SPEC <spec body>

<predicate body> ::= [IND <assertion>]
   | [PAR <varsort list>] DEF <assertion>

<function body> ::= [IND <assertion>]
   | [PAR <varsort list>] DEF <term>

<process body> ::= [[PAR <varsort list>] DEF <process expr>]

<set body> ::= [IND <assertion>]

<assertion> ::= TRUE
   | FALSE
   | <term>!
   | <term> = <term>
   | <predicate-name> <term list>
   | <set-name> <action term list>
   | NOT <assertion>
   | <assertion> ; <assertion>
   | <assertion> AND <assertion>
   | <assertion> OR <assertion>
   | <assertion> => <assertion>
   | <assertion> <=> <assertion>
   | FORALL <varsort list> <assertion>
   | EXISTS <varsort list> <assertion>
   | LET {<assignment> ','} ; <assertion>
   | ( <assertion> )

<comm assertion> ::= <action term> | <action term> = <action term>
   | <comm assertion> ; <comm assertion>
   | FORALL <varsort list> <comm assertion>
   | ( <comm assertion> )

<spec assertion> ::= <process-name> <term list> = <process expr>
   | <spec assertion> ; <spec assertion>
   | FORALL <varsort list> <spec assertion>
   | ( <spec assertion> )

<term> ::= <object-var>
   | <function-name> <term list>
   | THAT <varsort> <assertion>
   | LET {<assignment> ','} ; <term>
   | ( <term> )

<action term list> ::= {<action term> ','}
   | ( <action term list> )

<action term> ::= <action-name> <term list>
   | ( <action term> )

<term list> ::= {<term> ','}
   | ( <term list> )

<process expr> ::= PRETAU
   | DELTA
   | <process-name> <term list>
   | <process expr> . <process expr>
   | <process expr> + <process expr>
   | <process expr> || <process expr>
   | GCMD <ass-process expr>
   | SUM <varsort list> <process expr>
   | MERGE <varsort list> <process expr>
   | ENCAPS  <set-process expr>
   | HIDE <set-process expr>
   | ( <process expr> )

<set-process expr> ::= <set expr> , <process expr>
   | (<set-process expr>)

<ass-process expr> ::= <assertion>, <process expr>
   | (<ass-process expr>)

<set expr> ::= <set-name>
   | <set expr> + <set expr>
   | <set expr> & <set expr>
   | <set expr> ^ <set expr>
   | ( <set expr> )

<domain> ::=  {<sort-name> '#'}

<varsort list> ::= {<varsort> ','}

<varsort> ::= <object-var> : <sort-name>

<assignment> ::= <object-var> := <term>

<scheme-var> ::= <identifier>
```

```
<sort-name> ::= <identifier>

<predicate-name> ::= <identifier>

<function-name> ::= <identifier>

<action-name> ::= <identifier>

<process-name> ::= <identifier>

<set-name> ::= <identifier>

<object-var> ::= <identifier>
```

3.4 SDF Definition

Next, we give a definition of PSF/C in the Syntax Definition Formalism of HEERING & KLINT [HK89].
SDF 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 [Joh79] and YACC [LS79]. It is possible to generate a lexical scanner and some parse tables from such an SDF-definition [Rek87]. These parse tables together with a universal parser form a parser for the specified language. It is also possible to generate a so-called syntax directed editor from a description of the layout and the parse tables. This whole system is being implemented in LISP as part of ESPRIT Project 2177: GIPE (Generation of Interactive Programming Environments).

3.4.1 SDF Syntax

An SDF definition consists of two parts: a *lexical syntax* and a *context-free syntax*. In both parts we deal with the notions *sort* and *function* that correspond, respectively, to non-terminals and to production rules as used in BNF grammars [AU77].
This is an adaptation of an example of an SDF definition taken from [HK86].

```
module example
begin

  lexical  syntax

    sorts
      digit, letter, int, id, id-tail, comment-char

    layout
      white-space, comment

    functions
      [a-z]                   -> letter
      [0-9]                   -> digit
      digit+                  -> int
      [a-z0-9]                -> id-tail
      letter id-tail*         -> id
      [ \n\t\f\r]                -> white-space
      ~[{}]                   -> comment-char
      "{" comment-char* "}"      -> comment


  context-free  syntax

    sorts
      expr
    priorities
      "+" < "*"

    functions
      expr "+" expr    -> expr   {par, left-assoc}
      expr "*" expr    -> expr   {par, left-assoc}
      id               -> expr

end example
```

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

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

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

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

The *priorities* declaration is used to define the relative priority between functions. When unambiguous, the function may be abbreviated by its keyword skeleton. The associativity of functions may be declared by means of the attributes: *assoc, left-assoc* and *right-assoc* while the attribute *par* can be added to the function declaration to state that the function may be surrounded by parentheses in order to change its priority.

3.4.2 PSF/C in SDF

In this section we give the definition of the syntax of PSF/C using SDF.

```
module PSF/C
begin

  lexical syntax

    sorts
      id-char,identifier,
      comment-1-char, comment-2-char

    layout
      white-space, comment

    functions
      [0-9a-zA-Z"'/_]                          -> id-char
      id-char+                                 -> identifier

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

      ~ [\n]                                   -> comment-1-char
    ~ [}]                                                 -> comment-2-char
    "%" comment-1-char* "\n"                              -> comment
    "{" comment-2-char* "}"                               -> comment

context-free syntax

  sorts
    design, component, scheme, signature,
    item, class, definition, predicate-body, function-body,
    process-body, set-body, assertion, comm-assertion,
    spec-assertion, term, action-term, term-list,
    process-expr, set-process-expr, ass-process-expr, set-expr,
    domain, varsort-list, varsort, assignment, scheme-var,
    sort-name, predicate-name, function-name, action-name,
    process-name, set-name, object-var


  functions

    "DESIGN" {component ";"}* "SYSTEM" {scheme ","}*     -> design

    "COMP" scheme-var ":" scheme ":=" scheme             -> component
    "COMP" scheme-var ":" scheme                         -> component
    "LET" scheme-var ":=" scheme                         -> component

    class                                                -> scheme
    "IMPORT" scheme "INTO" scheme                        -> scheme
    "EXPORT" signature "FROM" scheme                     -> scheme
    "LET" scheme-var ":=" scheme ";" scheme              -> scheme
    scheme-var                                           -> scheme

    {item ","}*                                          -> signature
    signature "+" signature                              -> signature {left-assoc}
    item "^" signature                                   -> signature
    "SIG" scheme                                         -> signature

    "SORT" sort-name                                     -> item
    "PRED" predicate-name ":" domain                     -> item
    "FUNC" function-name ":" domain "->" sort-name       -> item
    "ACTION" action-name  ":" domain                     -> item
    "PROCESS" process-name  ":" domain                   -> item
    "SET" set-name                                       -> item

    "CLASS" definition* "END"                            -> class

    "SORT" sort-name                                     -> definition
    "PRED" predicate-name ":" domain predicate-body      -> definition
    "FUNC" function-name ":" domain "->"
                             sort-name function-body     -> definition
    "AXIOM" assertion                                    -> definition
    "ACTION" action-name ":" domain                      -> definition
    "PROCESS" process-name ":" domain  process-body      -> definition
    "SET" set-name set-body                              -> definition
    "COMM" comm-assertion                                -> definition
    "SPEC" spec-assertion                                -> definition

    "IND" assertion                                      -> predicate-body
    "PAR" varsort-list "DEF" assertion                   -> predicate-body
    "DEF" assertion                                      -> predicate-body
                                                         -> predicate-body

    "IND" assertion                                      -> function-body
```

```
"PAR" varsort-list "DEF" term                   -> function-body
"DEF" term                                      -> function-body
                                                -> function-body

"PAR" varsort-list "DEF" process-expr           -> process-body
"DEF" process-expr                              -> process-body
                                                -> process-body

"IND" assertion                                 -> set-body
                                                -> set-body

"TRUE"                                          -> assertion
"FALSE"                                         -> assertion
term "!"                                        -> assertion
term "=" term                                   -> assertion
predicate-name term-list                        -> assertion
set-name "[" action-term "]"                    -> assertion
"NOT" assertion                                 -> assertion
assertion ";" assertion                         -> assertion {left-assoc}
assertion "AND" assertion                       -> assertion {left-assoc}
assertion "OR" assertion                        -> assertion {left-assoc}
assertion "=>" assertion                        -> assertion {left-assoc}
assertion "<=>" assertion                       -> assertion {left-assoc}
"FORALL" varsort-list assertion                 -> assertion
"EXISTS" varsort-list assertion                 -> assertion
"LET" {assignment ","}* ";" assertion           -> assertion
"(" assertion ")"                               -> assertion {bracket}

action-term "|" action-term "=" action-term     -> comm-assertion
comm-assertion ";" comm-assertion               -> comm-assertion {left-asso
"FORALL" varsort-list comm-assertion            -> comm-assertion
"(" comm-assertion ")"                          -> comm-assertion {bracket}

process-name term-list "=" process-expr         -> spec-assertion
spec-assertion ";" spec-assertion               -> spec-assertion {left-asso
"FORALL" varsort-list spec-assertion            -> spec-assertion
"(" spec-assertion ")"                          -> spec-assertion {bracket}

object-var                                      -> term
function-name term-list                         -> term
"THAT" varsort assertion                        -> term
"LET" {assignment ","}* ";" term                -> term

action-name term-list                           -> action-term

"(" {term ","}+ ")"                             -> term-list {bracket}
                                                -> term-list

action-term                                     -> process-expr
"PRETAU"                                        -> process-expr
"DELTA"                                         -> process-expr
process-name term-list                          -> process-expr
process-expr "." process-expr                   -> process-expr
process-expr "+" process-expr                   -> process-expr
process-expr "||" process-expr                  -> process-expr
"GCMD" ass-process-expr                         -> process-expr
"SUM" sum-merge-arg                             -> process-expr
"MERGE" sum-merge-arg                           -> process-expr
"ENCAPS"  set-process-expr                      -> process-expr
"HIDE" set-process-expr                         -> process-expr
"(" process-expr ")"                            -> process-expr
```

```
    varsort-list "(" process-expr ")"                    -> sum-merge-arg

    "(" assertion "," process-expr ")"                   -> ass-process-expr

    "(" set-expr "," process-expr ")"                    -> set-process-expr

    set-name                                             -> set-expr
    set-expr "+" set-expr                                -> set-expr
    set-expr "&" set-expr                                -> set-expr
    set-expr "^" set-expr                                -> set-expr
    "(" set-expr ")"                                     -> set-expr

    {sort-name "#"}*                                     -> domain

    {varsort ","}*                                       -> varsort-list

    object-var ":" sort-name                             -> varsort

    object-var ":=" term                                 -> assignment

    identifier                                           -> scheme-var
    identifier                                           -> sort-name
    identifier                                           -> predicate-name
    identifier                                           -> function-name
    identifier                                           -> action-name
    identifier                                           -> process-name
    identifier                                           -> set-name
    identifier                                           -> object-var

end PSF/C
```

4 SEMANTICS

4.1 Introduction

The semantics of the COLD-K language can be found in [FJKR87]. That semantical model will be used as a base to define the semantics of PSF/C. All constructs in PSF/C that are already part of COLD-K have the same meaning as their counterparts in COLD-K. New constructs, i.e. all constructs dealing with process behaviour, are indirectly defined using the COLD-K semantics. This is done by giving a translation from PSF/C into COLD-K.
The intention is to give a semantics to the process definition part that resembles the algebraic semantics normally attached to process algebra (see e.g. BERGSTRA & KLOP [BK84, BK86b]). In order to be able to understand the formal translation, we will give an overview of the usual algebraic semantics for process algebra expressions.

4.2 ACP

We start from a given set A of atomic actions. Atomic actions are the simplest kind of processes, indivisible, and usually considered as having no duration. Complex processes can be constructed from simpler ones by applying several predefined functions and operators. Each atomic action is a constant in the set Action. The set Action is embedded in the set of processes, named Process.
On A, we have given a partial binary function γ, the *communication function*. γ must be commutative and associative, i.e.

$\gamma(a,b) = \gamma(b,a)$

$\gamma(a,\gamma(b,c)) = \gamma(\gamma(a,b),c)$

(when defined) for all $a,b,c \in A$. If $\gamma(a,b) = c$, we say a and b *communicate*, and the result of their communication is c. If $\gamma(a,b)$ is undefined, we say that a and b do not communicate. A and γ can be considered as *parameters* of the theory: in each application we will have to specify what atomic actions we have, and how they communicate. In PSF/C, we write $\gamma(a,b) = c$ as $a \mid b = c$.

On the domain of processes we define an equivalence relation by making a number of identifications between processes. These identifications follow from a set of axioms. For all processes x and y e.g. we consider the processes x+y and y+x to be identical. The intuition behind the identifications will be explained next.

The first two compositional operators we consider are $\cdot$, denoting sequential composition, and + for alternative composition. If x and y are two processes, then $x \cdot y$ is the process that starts the execution of y after the completion of x, and x+y is the process that chooses either x or y and executes the chosen process (not the other one). Each time a choice is made, we choose from a set of alternatives. We do not specify whether a choice is made by the process itself, or by the environment. Axioms A1-5 in table 1 below give the laws that + and $\cdot$ obey. We leave out $\cdot$ and brackets as in regular algebra, so $xy + z$ means $(x \cdot y) + z$. $\cdot$ will always bind stronger than other operators, and + will always bind weaker.

On intuitive grounds $x(y + z)$ and $xy + xz$ present different mechanisms (the moment of choice is different), and therefore, an axiom $x(y + z) = xy + xz$ is not included.

We have a special constant δ denoting deadlock, the acknowledgement of a process that it cannot do anything any more, the absence of any alternative. Axioms A6-7 give the laws for δ. We also have a special constant t that is used for *pre-abstraction* (see the following section). t or δ are not in the given set A, but are in the set of constants Action. Thus, γ is not defined for constants t, δ, which means that t or δ do not communicate.

Next, we have the parallel composition operator $\|$, called merge. The merge of processes x and y will interleave the actions of x and y, except for the communication actions. In $x \| y$, we can either do a step from x, or a step from y, or x and y both synchronously perform an action, which together make up a new action, the communication action. This trichotomy is expressed in axiom CM1. Here, we use two auxiliary operators $\mathbin{\|\!\_}$ (left-merge) and $\mid$ (communication merge). Thus, $x \mathbin{\|\!\_} y$ is $x \| y$, but with the restriction that the first step comes from x, and $x \mid y$ is $x \| y$ with a communication step as the first step. Axioms CM2-9 and CF1-2 give the laws for $\mathbin{\|\!\_}$ and $\mid$. The laws CF1-2, that say that on atomic actions $\mid$ coincides with γ, differ slightly from laws C1-3 in BERGSTRA & KLOP [BK84]. Finally, we have in table 1 the encapsulation operator ∂_H. Here H is a set of atomic actions ($H \subseteq A$), and ∂_H blocks those actions, renames them into δ. The operator ∂_H can be used to encapsulate a process, i.e. to block communications with the environment. Since $t \notin A$, always $\partial_H(t) = t$.

$x + y = y + x$		A1
$(x + y) + z = x + (y + z)$		A2
$x + x = x$		A3
$(x + y)z = xz + yz$		A4
$(xy)z = x(yz)$		A5
$x + \delta = x$		A6
$\delta x = \delta$		A7
$a \mid b = \gamma(a,b)$	if $\gamma(a,b)$ is defined	CF1
$a \mid b = \delta$	otherwise	CF2
$x \| y = x \mathbin{\|\!\_} y + y \mathbin{\|\!\_} x + x \mid y$		CM1
$a \mathbin{\|\!\_} x = ax$		CM2
$ax \mathbin{\|\!\_} y = a(x \| y)$		CM3
$(x + y) \mathbin{\|\!\_} z = x \mathbin{\|\!\_} z + y \mathbin{\|\!\_} z$		CM4

$a \mid bx = (a \mid b)x$	CM5
$ax \mid b = (a \mid b)x$	CM6
$ax \mid by = (a \mid b)(x \| y)$	CM7
$(x + y) \mid z = x \mid z + y \mid z$	CM8
$x \mid (y + z) = x \mid y + x \mid z$	CM9
$\partial_H(a) = a \quad \text{if } a \notin H$	D1
$\partial_H(a) = \delta \quad \text{if } a \in H$	D2
$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(xy) = \partial_H(x) \cdot \partial_H(y)$	D4

Table 1. ACP.

In this table, a,b $\in$ Action (= $A \cup \{t, \delta\}$), $H \subseteq A$, and x,y,z are arbitrary processes. In addition to the axioms of ACP, we often use the following axioms of Standard Concurrency.

$x \| \delta = x\delta = \delta \| x$	SC1
$(x \| y) \| z = x \| (y \| z)$	SC2
$x \| y = y \| x$	SC3

Table 2. Standard Concurrency.

4.3 Pre-abstraction

In system *verification*, it is essential that we can abstract from the internal actions of a system, in order to prove that the external behaviour is as specified beforehand. Here, we are defining a *specification* language, and we do not want to deal with silent steps, and a suitable set of axioms for such steps. Thus, we are dealing with *concrete* process algebra (process algebra without silent steps. A first (important) step in dealing with internal actions can however be made in concrete process algebra, and this is that we can give all internal actions the same name. We use the constant t for this purpose. The unary operator t_I will rename all atomic actions from the set I into t. We call the operator t_I *pre-abstraction* and we sometimes call the constant t *pre-tau*. These notions were introduced in BAETEN & BERGSTRA [BB88]. The axioms for t_I are presented in table 3.

$t_I(a) = a$	if $a \notin I$	PT1
$t_I(a) = t$	if $a \in I$	PT2
$t_I(x + y) = t_I(x) + t_I(y)$		PT3
$t_I(xy) = t_I(x) \cdot t_I(y)$		PT4

Table 3. Pre-abstraction.

4.4 Guarded command

We want to extend the axiom system ACP with generalized sum and generalized merge constructs. In order to do this, it is very useful to introduce the guarded command construct first. If ϕ is an assertion in MPL, and p is a process expression, we write $\phi :\rightarrow p$
for the process that is p if ϕ holds. If ϕ does not hold, we get deadlock. Notice that the advantage of the $\phi :\rightarrow p$ notation is exploited mainly in cases where ϕ (and p) contains occurences of free variables for data. It is easy to write down the axioms for the guarded command. See table 4. Here, and in the

following sections, we will use the common logical connectives ($\wedge, \vee, \neg$), which are defined for assertions in MPL.

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

Table 4. Guarded Command.

Example: we can define the *if...then...else* construction by:

if ϕ *then* p *else* $q = \phi :\rightarrow p + \neg\phi :\rightarrow q$.

4.5 Generalized sum and merge

In order to give some motivation for what is to follow, we discuss an example first. Consider a one-place buffer with one input port and two output ports, called O and E. Atomic actions are parameterized by natural numbers, elements of the data sort N. We have the actions in(n), outO(n) and outE(n) for each $n \in N$. The buffer will output all odd numbers received at port O, all even numbers at port E. A recursive equation for this buffer can be given as follows:

$$\mathrm{Buf} = \sum_{\substack{n \in N \\ n\ \mathrm{odd}}} \mathrm{in}(n) \cdot \mathrm{outO}(n) + \sum_{\substack{n \in N \\ n\ \mathrm{even}}} \mathrm{in}(n) \cdot \mathrm{outE}(n).$$

Now the advantage of the guarded command introduced in 4.4 is, that we can rewrite this as follows:

$$\mathrm{Buf} = \sum_{n \in N} (n\ \mathrm{odd}) :\rightarrow \mathrm{in}(n) \cdot \mathrm{outO}(n) + \sum_{n \in N} (n\ \mathrm{even}) :\rightarrow \mathrm{in}(n) \cdot \mathrm{outE}(n).$$

This makes that we need to describe the generalized sum and merge constructs with only two arguments: first, a list of variables with sort names, and second a process expression. If $\underline{x}$ is a list of variables, and $\underline{D}$ a list of sort names of same length, then we write $\underline{x} \in \underline{D}$ to denote that a variable in list $\underline{x}$ is an element of the corresponding sort name in list $\underline{D}$. Then, the form of the sum and merge constructs is as follows:

$$\sum_{\underline{x} \in \underline{D} \phi} p \qquad \text{respectively} \qquad \mathop{\Big\|}_{\underline{x} \in \underline{D} \phi} p,$$

where variables from $\underline{x}$ may occur in p.

Moreover we introduce as abbreviations:

$$\sum_{\underline{x}\in\underline{D}} p = \sum_{\underline{x}\in\underline{D},T} p \quad \text{and} \quad \Big\|_{\underline{x}\in\underline{D}} p = \Big\|_{\underline{x}\in\underline{D},T} p$$

The following holds:

$$\sum_{\underline{x}\in\underline{D},\phi} p = \sum_{\underline{x}\in\underline{D},T} \phi{:}\!\rightarrow p = \sum_{\underline{x}\in\underline{D}} \phi{:}\!\rightarrow p$$

$$(\exists\, \underline{x}\in\underline{D}\ \phi) \Rightarrow \Big\|_{\underline{x}\in\underline{D},\phi} p = \Big\|_{\underline{x}\in\underline{D},T} \phi{:}\!\rightarrow p = \Big\|_{\underline{x}\in\underline{D}} \phi{:}\!\rightarrow p$$

Axioms for these constructs are non-trivial, but giving axioms is facilitated by using the guarded command of the previous section. We give the sum axioms in table 5.

$\sum_{\underline{x}\in\underline{D}} p = \sum_{\underline{x}\in\underline{D}} \phi{:}\!\rightarrow p + \sum_{\underline{x}\in\underline{D}} \neg\phi{:}\!\rightarrow p$		SUBSUM
$\sum_{\underline{x}\in\underline{D}} (\underline{x}=\underline{t}){:}\!\rightarrow p = p[\underline{x}:=\underline{t}]$	if no $\underline{x}$ occurs free in $\underline{t}$	SINGSUM

Table 5. Generalized sum.

Actually, in the translation to COLD-K, to be presented in section 4.6, we will use a different axiomatization of generalized sum, one that is easier to code in COLD.
The axioms in table 5 are sufficient to prove that each *finite* sum behaves as repeated applications of alternative composition (in fact, only assertions of the form $\underline{x}:=\underline{t}$ are needed).
We give an example: suppose we have the booleans B with constants TRUE and FALSE. Then:

$$\begin{aligned} \sum_{x\in B} p(x) &= \sum_{x\in B} (x=\text{TRUE}) :\rightarrow p(x) + \sum_{x\in B} (x=\text{FALSE}) :\rightarrow p(x) && \text{(by SUBSUM)} \\ &= p(\text{TRUE}) + p(\text{FALSE}) && \text{(by SINGSUM)} \end{aligned}$$

A useful additional axiom is the following axiom, which we can call FLATSUM:

$\sum_{\underline{x}\in\underline{D}} p = p$	if no $\underline{x}$ occurs free in p	FLATSUM

Table 6.

In order to deal with *infinite* sums, we need two additional axioms: ACTSUM, that says that any action performed by a sum construct must be an action of one of its summands, and the axiom of extensionality EXT, that says that a process is determined by its summands. These axioms are presented in table 7.

$\sum_{\underline{x}\in\underline{D}} p = \sum_{\underline{x}\in\underline{D}} p + a \Rightarrow \exists \underline{x}\in\underline{D}\,(p = p + a)$		ACTSUM 1
$\sum_{\underline{x}\in\underline{D}} p = \sum_{\underline{x}\in\underline{D}} p + a{\cdot}r \Rightarrow \exists \underline{x}\in\underline{D}\,(p = p + a{\cdot}r)$	no $\underline{x}$ free in r	ACTSUM 2
$\forall a\in A\,(p = p + a \Leftrightarrow q = q + a) \wedge$ $\forall a\in A, \forall r\,(p = p + a{\cdot}r \Leftrightarrow q = q + a{\cdot}r) \Rightarrow p = q$		EXT

Table 7. Infinite sums, extensionality.

The axioms for finite merge are similar to the axioms in table 5. We give them in table 8. Notice that we can derive that each empty sum is equal to δ, and the empty merge is defined to be δ in table 8.
In order to deal with infinite merges, we can have an axiom similar to ACTSUM in table 7. We prefer, however, not to do this, since one may hold the viewpoint that infinite merges do not occur "in reality". In this viewpoint, each infinite merge will equal CHAOS. Our theory here will not make a choice one way or the other.

$\forall \underline{x}\in\underline{D}\, \neg\phi \Rightarrow \underset{\underline{x}\in\underline{D},\, \phi}{\Vert} p = \delta$	EMPTYMERGE
$\exists \underline{x}\in\underline{D}\, \phi\wedge\psi \wedge \exists \underline{x}\in\underline{D}\, \phi\wedge\neg\psi \Rightarrow \underset{\underline{x}\in\underline{D},\, \phi}{\Vert} p = \underset{\underline{x}\in\underline{D},\, \phi\wedge\psi}{\Vert} p \;\Vert\; \underset{\underline{x}\in\underline{D},\, \phi\wedge\neg\psi}{\Vert} p$	SUBMERGE
$\underset{\underline{x}\in\underline{D}, \underline{x}=\underline{t}}{\Vert} p = p[\underline{x}:=\underline{t}]$ if no $\underline{x}$ occurs free in $\underline{t}$	SINGMERGE

Table 8. Generalized merge.

4.6 Translation to COLD-K

In this section we will only give some ideas behind the translation of PSF/C into COLD-K. The full description can be found in [BBMV90] since this is too technical to be presented in full detail here.
The translation is based on the concrete syntax of PSF/C which is passed through recursively. Each class in PSF/C is translated into a class in COLD-K. Moreover there is a predefined class called *BASIC* that includes definitions for items that are used in the translation. This class contains two sorts: *Action* and *Process*, and functions that operate on elements of these sorts like alternative and sequential composition.

```
CLASS
  ...
  SORT Process
  SORT Action
  FUNC alt : Process # Process -> Process
  FUNC seq : Process # Process -> Process
  ...
END;
```

All objects from PSF/C are translated into COLD-K objects, e.g. a process is translated into a function of sort *Process*. In the example below the left shows part of the PSF/C specification and the right the translation into COLD-K.

```
CLASS                            CLASS
  ...                              ...
  ACTION r : D                     FUNC r : D -> Action
  PROCESS read                     FUNC read : -> Process
  ...                              ...
END;                             END;
```

The class *BASIC* also includes the axioms presented in sections 4.2 through 4.5. The translation of axiom A3 (x = x + x) in table 1 of section 4.2 is the following:

```
AXIOM FORALL x:Process (
  alt(x,x) = x
)
```

Using these axioms we impose the equality relation on the sort *Process* and thus the semantics of the processes is defined. In this way we have reused the semantics of COLD-K to define a semantics of PSF/C.

5 EXAMPLES

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

5.1 A Vending Machine

5.1.1 The Problem

In this first example, adapted from MAUW & VELTINK [MV89], we want to specify a vending machine that sells tea and coffee. In fact this is a very simple machine, for it only accepts two kinds of coins, 10c coins and 25c coins, it does not give any change and there are no buttons to choose between coffee or tea. The choice is determined by whichever coin is inserted.

5.1.2 The Implementation

In our example we have used just one class, called VENDING_MACHINE_AND_USERS, to specify the vending machine. Firstly, we define all atomic actions that occur in the specification. The atomic actions fall apart into three categories. These categories are the actions of the vending machine, the action of the customer and the actions that are the result of a communication between the customer and the vending machine. In the COMM section we define all possible pairs of actions that can communicate with each other and we specify what the resulting action will be. This implicitly implies that all communications that are not listed here are prohibited. Next we define a set of atomic actions called H. This set contains all atomic actions that are performed by either the machine or the customer. Its use will show up later on. After having defined the atomic actions and the communication function we are able to specify the processes. The first process is called VMCT and represents the vending machine. Initially it offers the choice of a insert_10c or a insert_25c action, after which it continues to serve tea or coffee. After having served a drink VMCT returns to its initial state. The two next processes define a customer who wants tea and a customer who wants coffee. The last process defines the combination of the three previously defined processes. The vending machine is operating in parallel with the customers, in this example it serves a Tea_User followed by a Coffee_User, in that specific order. The ENCAPS operator forbids the atomic actions listed in H to occur on their own and such forces communication.

5.1.3 The Specification

```
%
% A very simple vending machine with two users.
%

LET VENDING_MACHINE_AND_USERS :=
CLASS
  ACTION insert_10c       :
  ACTION accept_10c       :
  ACTION 10c_paid         :
  ACTION insert_25c       :
  ACTION accept_25c       :
  ACTION 25c_paid         :
  ACTION serve_tea        :
  ACTION take_tea         :
  ACTION tea_delivered    :
  ACTION serve_coffee     :
  ACTION take_coffee      :
  ACTION coffee_delivered :

  COMM
    insert_10c   | accept_10c  = 10c_paid;
    insert_25c   | accept_25c  = 25c_paid;
    serve_tea    | take_tea    = tea_delivered;
    serve_coffee | take_coffee = coffee_delivered

  SET  H
   IND
    H(insert_10c);
    H(accept_10c);
    H(insert_25c);
    H(accept_25c);
    H(serve_coffee);
    H(take_coffee);
    H(serve_tea);
    H(take_tea)

  PROCESS VMCT :
  DEF ((accept_10c . serve_tea) +
       (accept_25c . serve_coffee)) . VMCT;

  PROCESS Tea_User :
  DEF insert_10c . take_tea;

  PROCESS Coffee_User :
  DEF insert_25c . take_coffee;

  PROCESS System :
  DEF ENCAPS(H, VMCT || ( Tea_User . Coffee_User ))

END;

VENDING_MACHINE_AND_USERS
```

5.2 A Landing Control System

5.2.1 The Problem

In the next example, adapted from MAUW & VELTINK [MV88], 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. 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.

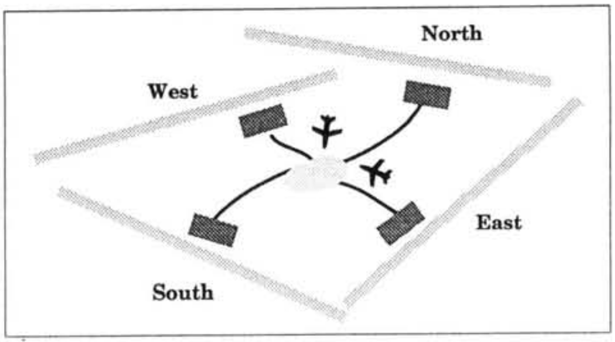

figure 1. Timbuktu Airport

5.2.2 The Implementation

The class *Landing_Control* is parameterized by the class *Airport*. This class 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 *Landing_Control* exports the atomic action *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 class 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.

5.2.3 The Specification

```
%
% Airport conditions local to Timbuktu-airport
%

LET TIMBUKTU_AIRPORT :=

CLASS
  SORT Strips
  SORT Plane_Ids
  FUNC North : -> Strips
  FUNC East  : -> Strips
  FUNC South : -> Strips
  FUNC West  : -> Strips
  FUNC KL204 : -> Plane_Ids
  FUNC SQ001 : -> Plane_Ids
  FUNC JL403 : -> Plane_Ids
  FUNC PA666 : -> Plane_Ids
  FUNC HA345 : -> Plane_Ids

END;


%
% The landing control system for Timbuktu airport.
%

LET TIMBUKTU_LANDING_CONTROL :=

EXPORT
  SORT Plane_Ids,
  ACTION receive_req_to_land : Plane_Ids,
  PROCESS Control :
FROM

IMPORT TIMBUKTU_AIRPORT INTO

CLASS
  ACTION receive_req_to_land : Plane_Ids
  ACTION read                : Strips # Plane_Ids
  ACTION send                : Strips # Plane_Ids
  ACTION communicate         : Strips # Plane_Ids
  ACTION land                : Strips # Plane_Ids
  ACTION disembark           : Plane_Ids

  COMM FORALL s:Strips, id:Plane_Ids
    (send(s,id) | read(s,id)  = communicate(s,id))

  SET  H
    IND FORALL s:Strips, id:Plane_Ids (
      H(read(s,id));
      H(send(s,id)) )

  PROCESS Distribution :
  DEF SUM id:Plane_Ids (receive_req_to_land(id) .
                        SUM s:Strips  (send(s,id))
                       ) . Distribution

  PROCESS Strip_Control : Strips
  PAR s:Strips
  DEF SUM id:Plane_Ids (read(s,id) . Handle(s,id)
                        ) . Strip_Control(s)

  PROCESS Handle : Strips # Plane_Ids
  PAR s:Strips, id:Plane_Ids
  DEF land(s,id) . disembark(id)

  PROCESS Control :
  DEF ENCAPS(H, Distribution ||
                MERGE s:Strips (Strip_Control(s))) )

END;
TIMBUKTU_LANDING_CONTROL
```

5.3 Alternating Bit Protocol

5.3.1 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 specification formalism. Our specification emanates from the ABP specification in ACP as described in BERGSTRA & KLOP [BK86a,BK86b].
We can represent the Alternating Bit Protocol with a picture as follows:

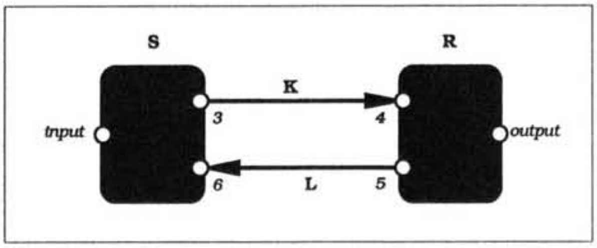

figure 2 Graphical representation of the Alternating Bit Protocol

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 will give a description of each component.

5.3.1.1 The Sender

First, component S reads a message at the input port. This message is extended with a *control boolean* to form a so-called *frame* and this frame is sent along channel K (3). The sending of the frame proceeds until component S receives an acknowledgement of a successful transmission at channel L (6). After a successful transmission component S flips the control boolean and starts all over again.

5.3.1.2 Communication Channel *K*

Component K transmits frames from the sender (3) to the receiver (4). 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.

5.3.1.3 The Receiver

The receiver R reads a frame from channel K (4). 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 (5) and waits for the retransmission of the frame.

5.3.1.4 Communication Channel *L*

Component *L* is used to transmit *receive acknowledgements* from the receiver (5) to the sender (6). 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.

5.3.2 The Implementation

The specification of the Alternating Bit Protocol starts of with some classes from the COLD IGLOO (Incremental Generic Library Of Objects), which has been collected at Philips Research, Eindhoven. These classes are BOOL_SPEC and FRAME_SPEC. The first one defines the booleans and the second one is a modification of TUP2_SPEC, which defines tuples of data types. In this case FRAME_SPEC defines a tuple of two booleans.
Next come the classes that are specific for this application. At first we have to model the unreliable channels of the protocol. Channels K and L are fairly similar, the only difference is that channel K transports frames while L transports booleans. There are three atomic actions involved with the definition of the unreliable channels: a read and a send action, and an error action indicating malfunctioning of the channel. The sender S and the receiver R are specified in SENDER_SPEC and RECEIVER_SPEC respectively.
Now that we have defined the separate objects of the system, we have to glue them together. This is done in the class ABP_SPEC. The specification of the sender, the receiver and the unreliable channels are imported and at the same time the resulting atoms of the communication between the several objects of the system are defined.
The last thing we have to do is to supply two objects, one at either side of the ABP process. One that supplies an infinite stream of random booleans, RANDOM_SPEC, and one that is able to read an infinite stream of random booleans, DRAIN_SPEC. In the final class ABP_SYSTEM_SPEC we tie together the RANDOM_SPEC, ABP_SPEC and DRAIN_SPEC.

5.3.3 The Specification

```
%
% Name : BOOL_SPEC
% Date : 09/03/88
%
% Description :
%
% This is a specification of the data type of booleans with
% inductive definitions for the non-constructor operations.

LET BOOL_SPEC :=

EXPORT

  SORT Bool,
  FUNC true  :               -> Bool,
  FUNC false :               -> Bool,
  FUNC not   : Bool          -> Bool,
  FUNC and   : Bool # Bool -> Bool,
  FUNC or    : Bool # Bool -> Bool,
  FUNC imp   : Bool # Bool -> Bool,
  FUNC eqv   : Bool # Bool -> Bool,
  FUNC xor   : Bool # Bool -> Bool

FROM
CLASS

  SORT Bool
  FUNC true  :-> Bool
  FUNC false :-> Bool
  AXIOM
  {BOOL1} true!;
  {BOOL2} false!;
  {BOOL3} NOT true = false

  PRED is_gen : Bool
  IND is_gen(true);
      is_gen(false)

  AXIOM FORALL b:Bool
  {BOOL4} is_gen(b)

  FUNC not: Bool -> Bool
  IND  not(true) = false;
       not(false) = true

  FUNC and: Bool # Bool -> Bool
  IND FORALL b:Bool
      ( and(false,b) = false;
        and(true,b) = b  )

  FUNC or: Bool # Bool -> Bool
  IND FORALL b:Bool
      ( or(false,b) = b;
        or(true,b) = true  )

  FUNC imp: Bool # Bool -> Bool
  IND FORALL b:Bool
      ( imp(false,b) = true;
        imp(true,b) = b  )

  FUNC eqv: Bool # Bool -> Bool
  IND FORALL b:Bool, c:Bool
      ( b = c        => eqv(b,c) = true;
        NOT  b = c  => eqv(b,c) = false )

  FUNC xor: Bool # Bool -> Bool
  IND FORALL b:Bool, c:Bool
      ( b = c        => xor(b,c) = false;
        NOT  b = c  => xor(b,c) = true )

END;


%
% Name : FRAME_SPEC
% Date : 10/03/88
%
% Description :
%
% This is an axiomatic specification of the 2-tuple data type
% with inductive definitions for the non-constructor operations.

LET FRAME_SPEC :=

EXPORT

  SORT Frame,
  SORT Bool,
  FUNC frame : Bool # Bool      -> Frame,
  FUNC proj1 : Frame            -> Bool,
  FUNC proj2 : Frame            -> Bool

FROM
IMPORT BOOL_SPEC INTO

CLASS
  SORT Frame   DEP Bool
  FUNC frame : Bool # Bool -> Frame

  AXIOM FORALL i1:Bool, j1:Bool, i2:Bool, j2:Bool (
  {TUP1} frame(i1,i2)!;
  {TUP2} frame(i1,i2) = frame(j1,j2) => i1 = j1 AND i2 = j2 )

  PRED is_gen: Frame
  IND FORALL i1:Bool, i2:Bool (
        is_gen(frame(i1,i2)) )

  AXIOM FORALL t:Frame
  {TUP3} is_gen(t)

  FUNC proj1: Frame -> Bool
  IND FORALL i1:Bool, i2:Bool (
        proj1(frame(i1,i2)) = i1 )

  FUNC proj2: Frame -> Bool
  IND FORALL i1:Bool, i2:Bool (
        proj2(frame(i1,i2)) = i2 )

END;


%
% This is a specification of an unreliable channel that
% either transports one item from its input to its output,
% or generates an error stating malfunctioning
%

LET UC_K_SPEC :=

EXPORT
  SORT Frame,
  PROCESS UC_K: ,
  ACTION K_read: Frame ,
  ACTION K_send: Frame ,
  ACTION K_error:
FROM

IMPORT FRAME_SPEC INTO

CLASS
  ACTION K_read: Frame
  ACTION K_send: Frame
  ACTION K_error:

  PROCESS UC_K:
  DEF SUM d:Frame (K_read(d) . UC_K(d));

  PROCESS UC_K: Frame
  PAR d:Frame
  DEF (skip . K_send(d) + skip . K_error) . UC_K

END;
```

```
%
% This is a specification of an unreliable channel that
% either transports one item from its input to its output,
% or generates an error stating malfunctioning
%

LET UC_L_SPEC :=

EXPORT
  SORT Bool,
  PROCESS UC_L: ,
  ACTION L_read: Bool ,
  ACTION L_send: Bool ,
  ACTION L_error:
FROM

IMPORT BOOL_SPEC INTO

CLASS
  ACTION L_read: Bool
  ACTION L_send: Bool
  ACTION L_error:

  PROCESS UC_L:
  DEF SUM d:Bool (L_read(d) . UC_L(d));

  PROCESS UC_L: Bool
  PAR d:Bool
  DEF (skip . L_send(d) + skip . L_error) . UC_L

END;


%
% This is a specification of the sender of the Alternating Bit Protocol
%

LET SENDER_SPEC :=

EXPORT
  SORT Frame,
  SORT Bool,
  PROCESS S : ,
  ACTION read_item: Bool ,
  ACTION send_frame: Frame ,
  ACTION read_ack: Bool ,
  ACTION read_ack_error:
FROM

IMPORT BOOL_SPEC INTO
IMPORT FRAME_SPEC INTO

CLASS
  ACTION read_item: Bool
  ACTION send_frame: Frame
  ACTION read_ack: Bool
  ACTION read_ack_error:

  PROCESS S :
  DEF RM(false)
  PROCESS RM : Bool
  PAR b:Bool
  DEF SUM d:Bool (read_item(d) . SF(d,b))

  PROCESS SF : Bool # Bool
  PAR d:Bool, b:Bool
  DEF send_frame(frame(d,b)) . RA(d,b)

  PROCESS RA : Bool # Bool
  PAR d:Bool, b:Bool
  DEF (read_ack(not(b)) + read_ack_error) . SF(d,b)
     + read_ack(b) . RM(not(b))

END;


%
% This is a specification of the receiver of the Alternating Bit Protocol
%

LET RECEIVER_SPEC :=

EXPORT
  SORT Frame,
  SORT Bool,
  SORT Bool,
  PROCESS R : ,
  ACTION send_item: Bool ,
  ACTION read_frame: Frame ,
  ACTION send_ack: Bool ,
  ACTION read_frame_error:
FROM

IMPORT FRAME_SPEC INTO
IMPORT BOOL_SPEC INTO

CLASS
  ACTION send_item: Bool
  ACTION read_frame: Frame
  ACTION send_ack: Bool
  ACTION read_frame_error:

  PROCESS R :
  DEF RF(false);

  PROCESS RF : Bool
  PAR b:Bool
  DEF (SUM d:Bool (read_frame(d,not(b))) + read_frame_error)
         . SA(not(b))
       + SUM d:Bool (read_frame(d,b) . SM(d,b))

  PROCESS SA : Bool
  DEF send_ack(b) . RF(not(b))

  PROCESS SM : Bool # Bool
  PAR d:Bool, b:Bool
  DEF send_item(d) . SA(b)

END;
```

```
%
% This is a specification of the Alternating Bit Protocol, which
% combines all previously defined classes into one system
%

LET ABP_SPEC :=

EXPORT

  SORT Bool,
  PROCESS ABP : ,
  ACTION read_item : Bool ,
  ACTION send_item : Bool

FROM

IMPORT BOOL_SPEC INTO
IMPORT FRAME_SPEC INTO
IMPORT UC_K_SPEC INTO
IMPORT UC_L_SPEC INTO
IMPORT SENDER_SPEC INTO
IMPORT RECEIVER_SPEC INTO


CLASS

  ACTION frame_error :
  ACTION ack_error :
  ACTION ack_enters_channel : Bool
  ACTION ack_leaves_channel : Bool
  ACTION frame_enters_channel : Frame
  ACTION frame_leaves_channel : Frame

COMM
  K_error | read_frame_error = frame_error;
  L_error | read_ack_error = ack_error

COMM FORALL b:Bool (
  send_ack(b) | L_read(b) = ack_enters_channel(b);
  L_send(b) | read_ack(b) = ack_leaves_channel(b) )

COMM FORALL f:Frame (
  send_frame(f) | K_read(f) = frame_enters_channel(f);
  K_send(f) | read_frame(f) = frame_leaves_channel(f) )

SET  H
  IND FORALL d:Bool, b:Bool, f:Frame (
    H(K_error);
    H(read_frame_error);
    H(L_error);
    H(read_ack_error);
    H(read_item(d));
    H(send_item(d));
    H(send_ack(b));
    H(read_ack(b));
    H(L_read(b));
    H(L_send(b));
    H(send_frame(f));
    H(K_read(f));
    H(read_frame(f));
    H(K_send(f)) )
  PROCESS ABP :
  DEF ENCAPS(H, S || R || UC_K || UC_L)

END;


%
% This is a specification of a process that produces a random stream
% of booleans
%

LET RANDOM_SPEC :=

EXPORT

  SORT Bool,
  PROCESS RANDOM : ,
  ACTION output : Bool

FROM

IMPORT BOOL_SPEC INTO

CLASS

  ACTION output : Bool
  PROCESS RANDOM :
  PAR d:Bool
  DEF SUM d:Bool (SKIP . output(d)) . RANDOM )

END;


%
% This is a specification of a process consuming booleans
%

LET DRAIN_SPEC :=

EXPORT

  SORT Bool,
  PROCESS DRAIN : ,
  ACTION input : Bool

FROM

IMPORT BOOL_SPEC INTO

CLASS

  ACTION input : Bool
  PROCESS DRAIN :
  PAR d:Bool
  DEF SUM d:Bool (input(d)) . DRAIN )

END;
```

```
%
% Here the total system is created by linking together the subsystems.
%

LET ABP_SYSTEM_SPEC :=

EXPORT
  PROCESS ABP_SYSTEM :
FROM

IMPORT ABP_SPEC INTO
IMPORT DRAIN_SPEC INTO
IMPORT RANDOM_SPEC INTO

CLASS
  ACTION item_read : Bool
  ACTION item_sent : Bool

COMM FORALL d:Bool (
  output(d) | read_item(d) = item_read(d);
  send_item(d) | input(d) = item_sent(d) )

SET  H
  IND FORALL d:Item (
    H(output(d));
    H(input(d));
    H(read_item(d));
    H(send_item(d)) )

  PROCESS ABP_SYSTEM :
  DEF ENCAPS(H, RANDOM || ABP || DRAIN)

END;

ABP_SYSTEM_SPEC
```

6 EXTENSIONS

A number of possible extensions of PSF/C come to mind, most of them concerning the modularization of classes and the addition of extra process composition operators. We mention a few of them.

Firstly, like in full COLD we could add parameterization concepts and renamings on the class level. Secondly, instead of having only two simple renaming operators, on the process specification level, viz. encapsulation (that renames a set of atomic actions into δ, leaving other actions fixed) and pre-abstraction (renaming into t), we can allow general *renaming operators*, having an operator ρ_f for each function f from A into the set Action. For more details, see BAETEN & BERGSTRA [BB88]. In this paper, also generalized renaming operators can be found, most notably the *state operator*, with which we can keep track of the state of a process during execution. This operator finds applications in the translation of programming languages or specification languages into process algebra.

Another issue is the addition of the silent step τ. This process is necessary for system verification. On the other hand, addition of a silent leads to complicated issues, one of which is the exact formulation of axioms. The concrete language ACP has remained fixed over a number of years, so is fairly well-established, and moreover is amenable to term rewriting analysis.

There are several other operators that can be added to PSF/C and will ease specifications. We can think of the *mode transfer operator*, the *priority operator*, determination of *alphabets*, *process creation* operator, etc.

The semantics of PSF/C can also be given in a different way than was presented here. Notably, it is possible to give an operational semantics with Plotkin-style rules, by defining a COLD predicate *arrow* on \Process # \Action # \Process, with all rule definitions translated into COLD axioms.

7 COMPARISON OF PSF/C WITH SIMILAR LANGUAGES

The most obvious candidate for comparison is PSF/ASF as it was described in [MV88]. The difference is that the data type specifications are now given in the way of COLD. Moreover the concrete syntax of the process declarations is formatted in the style of COLD. (In the case of PSF/ASF the process declarations were formatted in the style of ASF.) Because we wanted to use the data type specifications from COLD only the static fragment of it has been imported into PSF/C. It is an open question for us how the dynamic part of COLD could be combined with ACP. There seems to be an inherent overlap between the procedures in COLD and the processes of ACP. Due to this overlap an orthogonal language design based on a combination of COLD and ACP seems difficult to obtain.

The reason to consider a combination of ACP with COLD rather than with ASF is threefold:

(i) It is easier to base process declarations on data types specifed with first order formulae than on types that are algebraically specified using initial algebra semantics. Indeed for the precise definition of guardedness for systems of recursion equations negative information (i.e. information about expressions denoting different data) is essential. COLD allows the use of full first order specifications. The induction scheme of COLD also allows the restriction of data algebras to so-called minimal (term generated) algebras. So the expressive power exceeds that of ASF for all practical purposes. Of course there is a price to be paid: automatic specification and implementation of COLD specifications is not an easy matter. It is essentially harder than for the algebraic specifications of ASF

(ii) The major strong point of COLD is its modularisation mechanism. The power of that mechanism is already fully present in the static part. We observed that by simply adopting COLD for data type declaration, and using the same modularisation mechanisms also in the presence of process declarations one obtains a language for which a semantics can be defined in just the same way as for COLD. Indeed the meaning of PSF/C constructs is found by translating these into theories in the infinitary many sorted partial logic (as it was done in [FJKR 87]). For notational reasons this translation is found via an intermediate translation of PSF/C into COLD. It should be noted, however, that this mechanism can in principle be used to obtain a semantic description of PSF/ASF as well. That would require a meticulous and unpleasant translation of ASF into COLD however.

(iii) We are interested in the relation (and possible combinations) of COLD and ACP. It seems to be the obvious point of departure to begin with a language definition that combines COLD and ACP in the same way as LOTOS combines Act-one and CCS.

LOTOS (Language of Temporal Ordering Specification) [ISO87] is one of the two Formal Description Techniques, developed within ISO for the formal specification of open distributed systems, in particular for those related to the Open Systems Interconnection (OSI) computer architecture. Differences with PSF/C are: (i) bias towards CCS instead of ACP, (ii) COLD syntax is replaced by Act One, (iii) though modularisation concepts are available for the data type specifications, they are not for the process part as opposed to PSF/C where there is no distinction between data and process parts, and import/export constructs are offered for both. (iv) the semantics is given in terms of transition systems.

In MORELL MEERFORDT [Mor88], a syntactic combination of CSP and Meta IV is presented. The specification language is proposed and illustrated by examples. The main point is that processes can be parameterized by data structures. A systematic translation into Ada exists for this formalism.

(Differences with PSF/C: (i) bias towards CSP instead of bias towards ACP, (ii) there seems to have been paid less attention to modularisation, and of course (iii) COLD syntax is replaced by Meta IV. The difference between these formats is minimal for flat specifications, i.e. specifications without explicit modular structure.

No particular semantical model is selected to describe the semantics of the CSP/Meta IV combination. Probably the author has transition systems in mind.

ASTESIANO, MASCARI, REGGIO & WIRSING [AMRW85] describes the formalism SMOLCS for specifying concurrent systems. Differences with PSF/C are the following: (i) SMOLCS is biased towards CCS rather than to ACP, the semantics is presented in terms of transition systems (ii) although SMOLCS uses an algebraic formalism for data type specification (as does PSF/ASF fròm [MV88]) the semantic intuition is quite different because SMOLCS inherits the orientation towards hierarchical specifications that was proposed by the Munich School.

Although not apparent from the syntax one might say that SMOLCS is closer to LOTOS than to PSF/C.

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

8 Conclusion

In the construction of the language PSF/C, the design objectives stated in the introduction have been met. A few additional remarks:

- we found that the translation of the process constructions to COLD is cumbersome, and it is our preliminary conclusion that the resulting insights do not justify the effort.
- the SDF system suffices to generate simple tools for the language;
- we obtained a COLD oriented language in which certain comparative advantages of COLD over ASF are preserved. Thus, PSF/C has greater expressive power than PSF/ASF, and a more flexible semantic theory;

9 References

[AMRW85] E.Astesiano, G.F.Mascari, G.Reggio, M.Wirsing, *On the parametrised algebraic specification of concurrent systems*, Proc. 10th Colloquium on Trees in Algebra and Programming (TAPSOFT), LNCS 185, pp. 342-358, Springer Verlag, 1985.

[AU77] A.V. Aho & J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.

[BB88] J.C.M. Baeten & J.A. Bergstra, *Global renaming operators in concrete process algebra*, Inf. & Comp. 78 (3), 1988, pp. 205-245.

[BBMV90] J.C.M. Baeten, J.A. Bergstra, S. Mauw & G.J. Veltink, *A process specification formalism based on static COLD*, report P8906b, Programming Research Group, University of Amsterdam 1990.

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

[BK84] J.A. Bergstra & J.W. Klop, *Process algebra for synchronous communication*, Information & Control 60, 1984, pp. 109-137.

[BK86a] J.A. Bergstra & J.W. Klop, *Verification of an alternating bit protocol by means of process algebra,* in: Math. Methods of Spec. & Synthesis of Software Systems '85, (W. Bibel & K.P. Jantke, eds.), Math. Research 31, Akademie-Verlag Berlin, pp 9-23, 1986.

[BK86b] J.A. Bergstra & J.W. Klop, *Process algebra: specification and verification in bisimulation semantics,* in: Math. & Comp. Sci. II, (M. Hazewinkel, J.K. Lenstra & L.G.L.T. Meertens, eds.), CWI Monograph 4, pp 61-94, North-Holland, Amsterdam, 1986.

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

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

[HK86] J. Heering & P. Klint, *A syntax definition formalism,* Report CS-R8633, Centre for Mathematics and Computer Science, Amsterdam, 1986.

[HK89] J. Heering & P. Klint, *A syntax definition formalism,* in [BHK89], pp. 283-298.

[ISO87] International Organization for Standardization, *Information processing systems - Open systems interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour,* ISO/TC 97/SC 21, (E. Brinksma, ed.), 1987.

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

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

[Mor88] H. Morell Meerfordt, *Combining CSP and Meta IV into an Ada Related PDL for developing Concurrent Programs,* in: Ada in Industry, The Ada companion series (S. Heilbrunner, ed.), Cambridge University Press, pp. 157-171, 1988.

[MV88] S. Mauw & G.J. Veltink, *A process specification formalism,* report P8814, Programming Research Group, University of Amsterdam 1988.

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

[RdL89] G.R. Renardel de Lavalette, *COLD-K^2 , the static kernel of COLD-K,* Report RP/mod-89/8, Software Engineering Research Centrum, Utrecht, 1989.

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

[WB89] M. Wirsing, J.A. Bergstra, eds. , *The Design Language COLD,* section II in: Algebraic Methods: Theory, Tools and Applications, LNCS 394, pp. 139-328, Springer Verlag, 1989.