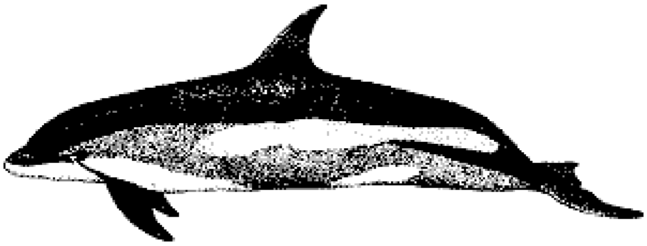
University of Amsterdam
Programming Research Group



Basic Multi-Competence Programming

Jan A. Bergstra
Inge Bethke
Alban Ponse

Report P0001 januari 2000

University of Amsterdam
Department of Computer Science
Programming Research Group

Basic Multi-Competence Programming

Jan A. Bergstra
Inge Bethke
Alban Ponse

Report P0001 januari 2000

J.A. Bergstra

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7591
e-mail: janb@wins.uva.nl

Department of Philosophy
Utrecht University

P.O.Box 80126
NL-3508 TC Utrecht
The Netherlands

tel. +31 30 253 5579
e-mail: Jan.Bergstra@phil.uu.nl

I. Bethke

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7583
e-mail: inge@wins.uva.nl

A. Ponse

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7592
e-mail: alban@wins.uva.nl

Universiteit van Amsterdam, 2000

Basic Multi-Competence Programming

Jan A. Bergstra[1,2] Inge Bethke[1] Alban Ponse[1]

[1] *University of Amsterdam, Programming Research Group*
`http://www.wins.uva.nl/research/prog/`

[2] *Utrecht University, Department of Philosophy*
`http://www.phil.uu.nl/eng/home.html`

Version 2.0

Abstract. BMCP (Basic Multi-Competence Programming) is a paradigm for teamwork for simple programming tasks. BMCP is simplified in the sense that programming tasks require no more than implementing a given functional specification. BMCP uses CVN (Conditional Value Notation) as a format for its functional specifications. BMCP displays seven competences, this collection being derived from an underlying family of four so-called program assessment perspectives. BMCP requires fixed agreements on interfacing, input and output. This version (2.0) replaces the earlier version 1.0 of this document.

Contents

1 Introduction

Multi-Competence Programming (MCP) is a programming framework meant for larger teams. Its objective is to get a firm grip on quality control supplementary to, and as much as possible simultaneously with, program construction. MCP is primarily an educational framework, delineated along 17 fresh acronyms. Taking the program algebra (see [5]) definition of what a program is as a point of departure, four *program assessment perspectives* emerge, each of which gives rise to a quality control *competence*. In terms of program algebra, a program is viewed as a text. The algorithmic content of this text is to be found via a translation (projection) to a text in a low level language (e.g., PGLA from [5]). The latter text can be viewed as a sequence of elementary instructions.

According to this view a program is a description of a sequence of instructions, the description perhaps given in a high level language which must be projected down to a low level notation in order to determine its (algorithmic) meaning. This view mainly matters if semantic issues are at stake. When programming in a known program notation, all semantic issues having been resolved already, the program algebra view must be complemented with other views of a pragmatic nature. Four of these views, or program assessment perspectives, are

the *black box test* perspective,
the *white box test* perspective,
the *white box review* perspective, and
the *white box verification* perspective.

Each of these perspectives gives rise to a competence. The competence refers to the (proven, accredited) ability to assess software from that perspective. It also refers to a role that a person or team can play in a software development activity. We get then the following acronyms, referring to the perspectives mentioned above:

BBTC:	Black Box Test Competence
WBTC:	White Box Test Competence
WBRC:	White Box Review Competence
WBVC:	White Box Verification Competence

These four perspectives and competences are further discussed in Sections 2 and 3. In a program development project based on MCP each of these competences is put in the hands of a different team. Because these competences are so totally different in practice, each having its own expert community, journals, guru's etc., we use the phrase multi-competence programming for a project planning involving all or most of these.

Having established the quality control related competences, three more competences are needed to keep a project up and running:

PCC:	Program Construction Competence
PMC:	Project Management Competence
KMC:	Knowledge Management Competence

Altogether seven competences emerge. The teams responsible for these competences are denoted with:

PCT:	Program Construction Team
PMT:	Program Management Team
KMT:	Knowledge Management Team
BBTT:	Black Box Test Team
WBTT:	White Box Test Team
WBRT:	White Box Review Team
WBVT:	White Box Verification Team

The MCP method has immediate implications for a class room setting. When a group of students is constructing programs in order to develop and/or demonstrate their programming abilities, the assessment of this output requires some care. In fact each student ought to be role aware. When produced programs are judged it must be known:

- from what perspective(s) this judgement will be made;
- how the persons responsible for making the judgement will approach their task.

MCP in its simplest form poses a programming problem in terms of a functional specification that has to be implemented. The program construction activity produces *proposed implementations*. Only the management role can upgrade a proposed implementation to the level of an implementation. This will always require some form of quality control. All major perspectives on product oriented (not: process oriented) quality control are represented in the scheme of four perspectives discussed above.

MCP uses CVN (Conditional Value Notation) as its format for functional specifications. This notation is introduced in Section 4. The advantages of CVN are:

- very concise and compact (though limited in its usability),
- captures the main error conditions: abrupt program abortion (leading to M, the meaningless state) and program divergence (leading to D, the divergent state),
- it uses conditions and quantifiers in a form which easily deals with D and M as mentioned above.

When programming tasks are simplified to the implementation of a given functional specification, avoiding any feedback to the specification itself, the framework is termed BMCP, Basic MCP. Otherwise it is called AMCP for Adaptive MCP. AMCP views the production of the functional specification as a competence as well. The specification production goes through various repetitions, including a range of program construction and assessment activities, which in turn lead to a next phase of the specification. This involves yet another competence, the *requirements analysis competence*. AMCP can only take place in a significantly more involved setting where the programs are used to support in reaching further objectives.

BMCP is considered a useful "educational" approximation of AMCP, even if BMCP is not so practical because AMCP happens to occur more in industrial projects. As an educational paradigm we first concentrate on BMCP and only thereafter consider projects involving AMCP. BMCP contains the bare essentials of programming and of programming related quality control.

There is a huge literature on software management describing numerous tools, techniques, methodologies and standards. We mention [1], [7], [9], [10], [13] and [18].

2 Four Perspectives on a Program

The description of 'what a program is' in program algebra [5] by no means exhausts the valid perspectives one may have on programs. Accepting the viewpoint that a program is a text which is given algorithmic content by a projection into a low level program notation, still four different perspectives on a program can be determined:

Black box (behavioral) perspective. A program is a black box (e.g., a compiled binary, not a "Black box", which is orange). All one can do is to run it and to perform black box tests to assess its behavior. (See e.g., [17].)

White box (behavioral) perspective. Program P is a known text. Properties of P are taken from its specification. These properties are assessed by means of black box tests and white box tests. White box tests take advantage of knowledge of the program text in order to spot difficulties more efficiently than in black box (random) mode.

White box literary perspective. The program text is to be assessed by means of informal but systematic judgement of the text (code walkthrough etc., see e.g., [12]).

White box verification perspective. The program is a text characterized by properties which have to be demonstrated in a rigorous (formal) fashion. Preferably, proofs are machine readable and machine checked. (Cf. [2].)

Each of these has its value in practice. Some implications of these perspectives are the following activities, mainly used for quality control:

Black box testing. Black box tests are generated from the specification (possibly automatically). A test environment is produced in which many tests can be run (automatically) on any proposed implementation of the specification. It is essential that proposed implementations strictly adhere to boundary conditions regarding interfaces.

White box testing. White box test can often also be generated automatically. It is important that adequate coverage is achieved, i.e., as much as possible all branches of the program must be visited by at least one test run.

Program verification. Proof generation and proof checking (if possible computer supported or automatically).

White box critical review. Here the program is understood as a pseudocode containing the solution of a problem (i.e., how to best implement the specification). In this perspective the fundamental merits of the proposed solution are considered. Speed of calculation, memory usage, clarity, independence of problematic language features, exploitation of interesting language/machine options etc.

3 Competence partitioning in BMC

In a BMC project seven different roles are to be distinguished. These roles are performed by a team. A team may consist of just one person, it is even possible that a person performs the work of different teams at the same time. The following teams can exist:

KMT (the knowledge management team) investigates the following questions:

- What kind of algorithm can be used; what can be expected/achieved?
- Which program constructions will be needed?
- What kind of verification techniques can used? how difficult will this be?
- How can BB/WB tests be derived in the case at hand?

KMT will not only produce some rationale for a proposed implementation but also a context in which the level of ambition of the proposal can be assessed.

PCT (the program construction team) generates a program and applies modifications if the interaction with any of the other teams makes that plausible.

BBTT (the black box test team) produces test beds for proposed implementations (given the specification of the required functionality).

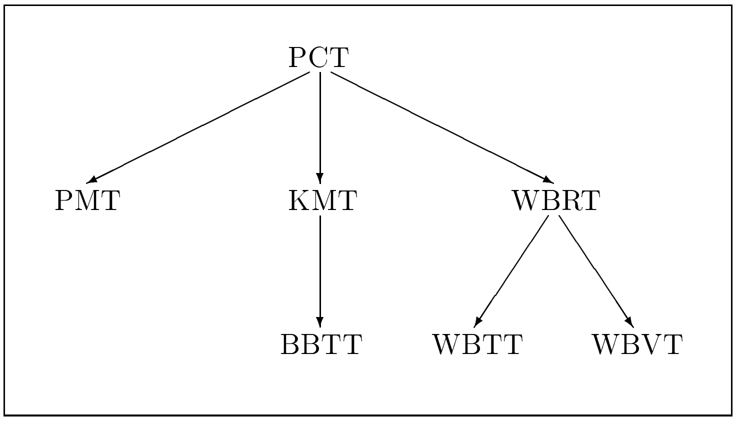
WBTT (the white box test team) delivers white box tests, preferably also in a largely automated setting.

WBRT (the white box review team) produces assessment of proposed implementations based on code inspections of various kinds.

WBVT (the white box verification team) produces correctness proofs.

PMT (the project management team) surveys the entire operation and manufactures the final deliverable containing programs as well as results of BBTT, WBTT, WBRT, WBVT, and KMT.

Amongst these teams, we discern the following hierarchy:



As suggested above, PCT takes the lead. PCT members intend to own the software. PCT uses KMT to produce background knowledge. KMT can manage BBTT as this requires specification expertise only. PCT uses WBRT for permanent assessment. WBRT in turn will guide both WBTT and WBVT (using insights of KMT). PCT uses the service of PMT to obtain better overall performance. For this reason PMT needs some power. This power is handed over by PCT in order to optimize PMT's role.

4 Conditional Value Notation

CVN, Conditional Value Notation, is a simple set of conventions to denote data values in the presence of possible errors. For each data type (data domain) V the presence of two exceptional values is assumed:

> D, a divergent value (the value of a diverging computation);
>
> M, a meaningless value (the value of an aborting computation).

Data values different from these exceptional values are called *good*. The predicate $g(\_)$ determines the good objects, i.e., $g(X)$ "holds" for some X in V if X is not D or M.[1] Thus

$$V = V_g \cup \{D, M\}$$

where $V_g = \{X \in V \mid g(X) \text{ holds}\}$. On V the symbol $=$ denotes equality, and $\neq$ denotes non-equality. With the special exception values D and M it often is simpler to use an asymmetric type of equality, called *left sequential equality* with notation $\mathrel{\underset{\circ}{=}}$. The idea is that $X \mathrel{\underset{\circ}{=}} Y$ gets its meaning by interpretation from left to right. For example, $M \mathrel{\underset{\circ}{=}} X$ equals M because equating the meaningless value with anything else is "meaningless". On $V = V_g \cup \{D, M\}$ left sequential equality is defined by:

$$\boxed{\begin{array}{lcll}
D \mathrel{\underset{\circ}{=}} X & = & D & \\
M \mathrel{\underset{\circ}{=}} X & = & M & \\
X \mathrel{\underset{\circ}{=}} D & = & D & \text{if } g(X) \text{ holds} \\
X \mathrel{\underset{\circ}{=}} M & = & M & \text{if } g(X) \text{ holds} \\
X \mathrel{\underset{\circ}{=}} X & = & T & \text{if } g(X) \text{ holds} \\
X \mathrel{\underset{\circ}{=}} Y & = & F & \text{if } g(X) \text{ and } g(Y) \text{ hold, and } X \neq Y
\end{array}}$$

[1] Instead of regarding a predicate as determining a subset, one might consider it as a function to $\{T, F\}$ and set $g(X) = T$ is $g(X)$ holds, and $g(X) = F$ otherwise.

For $X \underset{\circ}{=} Y = F$ we also write $X \underset{\circ}{\neq} Y$. Furthermore, in the case that other relations are defined over V_g, these can always be extended to V (including the error values) in a similar way. For example, in the domain $\mathbb{N}$ of the natural numbers the relation $<$ ("smaller than") can be extended to a relation $\circ\!\!<$ over $V = \mathbb{N} \cup \{D, M\}$ as follows:

$$\boxed{\begin{array}{lcll}
D \circ\!\!< X & = & D & \\
M \circ\!\!< X & = & M & \\
X \circ\!\!< D & = & D & \text{if } g(X) \text{ holds} \\
X \circ\!\!< M & = & M & \text{if } g(X) \text{ holds} \\
X \circ\!\!< Y & = & T & \text{if } g(X) \text{ and } g(Y) \text{ hold, and } X < Y \\
X \circ\!\!< Y & = & F & \text{if } g(X) \text{ and } g(Y) \text{ hold, and } X \geq Y
\end{array}}$$

It is evident how addition $(+)$ should be extended to $\circ\!\!+$ in $\mathbb{N} \cup \{D, M\}$. It is perhaps less evident how to extend subtraction $(-)$ to $\circ\!\!-$ in $\mathbb{N} \cup \{D, M\}$. Here the typical difference with $\circ\!\!+$ is caused by subtraction being a *partial* function on $\mathbb{N}$, i.e., $X - Y$ is not defined if $X < Y$. This problem is solved in $\mathbb{N} \cup \{D, M\}$ by the following equations:

$$\begin{array}{ll}
X \circ\!\!- Y = M & \text{if } g(X) \text{ and } g(Y) \text{ hold, and } X < Y \\
X \circ\!\!- Y = X - Y & \text{if } g(X) \text{ and } g(Y) \text{ hold, and } X \geq Y.
\end{array}$$

Consequently, $\circ\!\!-$ is not a partial function on $\mathbb{N} \cup \{D, M\}$.

As a convention we shall further use the common relation and operation symbols if we are sure that their arguments cannot be error values. (However, as the last example showed, the result value can be an error.)

Exercises.

1. Spell out the complete definition for subtraction on $\mathbb{N} \cup \{D, M\}$.

2. A different view on subtraction on $\mathbb{N}$ is to consider this operation as a partial one that may *diverge*. The idea is that

   $$psub(X, Y) = \textbf{if } X \underset{\circ}{=} Y \textbf{ then } 0 \textbf{ else } S(psub(X, S(Y))).$$

   (This example is taken from [3].) Here S is the *successor* function: $S(X) = X + 1$. Thus $psub(X, Y) = X - Y$ if $X \geq Y$. Describe how $psub$ can be defined on $\mathbb{N} \cup \{D, M\}$ if we set $S(D) = D$ and $S(M) = M$.

Four-Valued Conditions. In particular for the truth values T and F, instead of working with $V = \{T, F\}$ CVN proposes the use of $\mathbb{T}_4 = \{T, F, D, M\}$ (notations from [4]). We shortly elaborate on this setting, as to introduce the underlying logic over $\mathbb{T}_4$ that is useful for CVN. The logical connectives relevant for CVN are:

$\neg$	(negation),
$\mathbin{{}_{\circ}\!\wedge}$	(left sequential conjunction),
$\mathbin{{}^{\circ}\!\vee}$	(left sequential disjunction),
$\mathrel{\circ\!\!\rightarrow}$	(left sequential implication).

The (binary) connectives all have *right sequential* counterparts, e.g., $\mathbin{\wedge\!{}_{\circ}}$ is called right sequential conjunction and is defined by $X \mathbin{\wedge\!{}_{\circ}} Y = Y \mathbin{{}_{\circ}\!\wedge} X$.

So-called *truth tables* for $\neg$, $\mathbin{{}_{\circ}\!\wedge}$ and $\mathbin{{}^{\circ}\!\vee}$ are

X	$\neg X$
M	M
T	F
F	T
D	D

$\mathbin{{}_{\circ}\!\wedge}$	M	T	F	D
M	M	M	M	M
T	M	T	F	D
F	F	F	F	F
D	D	D	D	D

$\mathbin{{}^{\circ}\!\vee}$	M	T	F	D
M	M	M	M	M
T	T	T	T	T
F	M	T	F	D
D	D	D	D	D

These tables provide the definition of the logical connectives in $\mathbb{T}_4$, e.g., $M \mathbin{{}_{\circ}\!\wedge} F = M$ and $F \mathbin{{}_{\circ}\!\wedge} M = F$. Notice that $\mathbin{{}^{\circ}\!\vee}$ can be defined by *dualization*:

$$X \mathbin{{}^{\circ}\!\vee} Y = \neg(\neg X \mathbin{{}_{\circ}\!\wedge} \neg Y),$$

where the "=" is plain equality over $\mathbb{T}_4$. In this style the connective $\mathrel{\circ\!\!\rightarrow}$ is defined by $X \mathrel{\circ\!\!\rightarrow} Y = \neg X \mathbin{{}^{\circ}\!\vee} Y$. As another example of the use of this logic consider $\mathbb{N} \cup \{D, M\}$: the relation $\mathrel{{}_{\circ}\!\geq}$ can now be defined by

$$X \mathrel{{}_{\circ}\!\geq} Y = \neg(X \mathrel{\circ\!\!<} Y).$$

Exercises.

3. Determine the values of
   (a) $T \mathrel{\circ\!\!\rightarrow} (M \mathbin{{}^{\circ}\!\vee} T)$.
   (b) $(T \mathbin{{}_{\circ}\!\wedge} M) \mathbin{{}^{\circ}\!\vee} D)$.
   (c) $T \mathbin{{}_{\circ}\!\wedge} \neg(M \mathbin{{}^{\circ}\!\vee} F)$.

4. Show that the connectives $\mathbin{{}_{\circ}\!\wedge}$ and $\mathbin{{}^{\circ}\!\vee}$ are *associative*, i.e. $X \mathbin{{}_{\circ}\!\wedge} (Y \mathbin{{}_{\circ}\!\wedge} Z) = (X \mathbin{{}_{\circ}\!\wedge} Y) \mathbin{{}_{\circ}\!\wedge} Z$, and $X \mathbin{{}^{\circ}\!\vee} (Y \mathbin{{}^{\circ}\!\vee} Z) = (X \mathbin{{}^{\circ}\!\vee} Y) \mathbin{{}_{\circ}\!\wedge} Z$. (So we need not use brackets in repeated applications of each of these connectives.)

5. Let $V = V_g \cup \{D, M\}$ and consider the predicate g as a function from V to $\{T, F\}$. Let furthermore equality $(=)$ be given on V_g. Argue whether the expression

$$(\neg g(X) \mathbin{{}_{\circ}\!\wedge} X) \mathbin{{}^{\circ}\!\vee} (g(X) \mathbin{{}_{\circ}\!\wedge} \neg g(Y) \mathbin{{}_{\circ}\!\wedge} Y) \mathbin{{}^{\circ}\!\vee} (X = Y)$$

   defines left-sequential equality.

If_then and If_else. For the domain V (containing D, M) two operators are proposed by CVN:

$$p \cdot\!\!\rightarrow X \quad (p \text{ a four-valued condition, } X \in V)$$

$$X \nrightarrow Y \quad (X, Y \in V)$$

The expression $p \cdot\!\!\rightarrow X$ is pronounced "if P then X", and $X \nrightarrow Y$ is pronounced "if X else Y".

The operators $\cdot\!\!\rightarrow$ and $\nrightarrow$ are determined by the following equations:

$$\boxed{\begin{array}{rcl} T \cdot\!\!\rightarrow X & = & X \\ F \cdot\!\!\rightarrow X & = & M \\ D \cdot\!\!\rightarrow X & = & D \\ M \cdot\!\!\rightarrow X & = & M \end{array}}$$

and

$$\boxed{\begin{array}{rcl} X \nrightarrow Y & = & X \ \text{ if } g(X) \text{ holds} \\ D \nrightarrow X & = & D \\ M \nrightarrow X & = & X \end{array}}$$

So, $X \nrightarrow Y$ equals X if X is a good value or if X diverges, otherwise it equals Y. We notice that $\nrightarrow$ is *associative*, i.e.

$$X \nrightarrow (Y \nrightarrow Z) = (X \nrightarrow Y) \nrightarrow Z.$$

Therefore we can omit parentheses in repeated applications. Furthermore, to enhance readability we assume that $\mathbin{{}_{\circ}\!\wedge}$, $\mathbin{{}^{\circ}\!\vee}$ bind stronger than $\mathbin{\cdot\!\!\rightarrow}$, which in turn binds stronger than $\nrightarrow$, so

$$\neg p \mathbin{{}_{\circ}\!\wedge} q \mathbin{\cdot\!\!\rightarrow} X \nrightarrow Y = ((\neg p) \mathbin{{}_{\circ}\!\wedge} q) \mathbin{\cdot\!\!\rightarrow} X) \nrightarrow Y,$$

and the brackets in $p \mathbin{\cdot\!\!\rightarrow} (X \nrightarrow Y)$ cannot be omitted.

The following identities follow easily by considering all possible values of p and q, and will be used in the sequel:

1. $p \mathbin{\cdot\!\!\rightarrow} (X \nrightarrow Y) = p \mathbin{\cdot\!\!\rightarrow} X \nrightarrow p \mathbin{\cdot\!\!\rightarrow} Y$,
2. $p \mathbin{\cdot\!\!\rightarrow} (q \mathbin{\cdot\!\!\rightarrow} X) = p \mathbin{{}_{\circ}\!\wedge} q \mathbin{\cdot\!\!\rightarrow} X$.

With the operators $\mathbin{\cdot\!\!\rightarrow}$ and $\nrightarrow$, a two-sided conditional can be found as follows:

$$X \triangleleft p \triangleright Y = p \mathbin{\cdot\!\!\rightarrow} X \nrightarrow \neg p \mathbin{\cdot\!\!\rightarrow} Y.$$

(Cf. the notation **if** p **then** X **else** Y.)

Let V be given, and let $\Phi : V_g \to \mathbb{T}_4$ be a predicate on V_g (taking errors into account). There are two *quantifiers* that we want to use in CVN:

$$\forall X \in V_g(\Phi(X)) \quad (\text{“}\Phi(X) \text{ for all } X \in V_g\text{”})$$
$$\exists X \in V_g(\Phi(X)) \quad (\text{“there exists } X \in V_g \text{ for which } \Phi(X)\text{”})$$

We describe their values as follows:

$\forall X \in V_g(\Phi(X))$: if $\Phi(X) = T$ for all $X \in V_g$, then T,
else, if $\Phi(X) = M$ for some $X \in V_g$, then M,
else, if $\Phi(X) = F$ for some $X \in V_g$, then F,
else, D.

$\exists X \in V_g(\Phi(X))$: if $\Phi(X) = M$ for some $X \in V_g$, then M,
else, if $\Phi(X) = T$ for some $X \in V_g$, then T,
else, if $\Phi(X) = D$ for some $X \in V_g$, then D,
else, F.

Notice that quantification ranges over V_g, and that

$$\forall X \in V_g(\Phi(X)) = \neg \exists X \in V_g(\neg \Phi(X)).$$

Furthermore we shall sometimes use ‘bounded’ quantification over V_g. As an example, let $V_g = \mathbb{N}$ and $i \in \mathbb{N}$, then

$$\exists j \; 0 \leq j < i(\Phi(j))$$

expresses that there is a value j such that $0 \leq j$, $j < i$ and $\Phi(j)$ holds. (This is not a real extension: use of bounded quantification can be avoided at cost of readability).

The strength of CVN is its explicit dealing with the two fundamental error cases: abrupt abortion of a computation (leading to M) and computational divergence (modeled by D). CVN can be used to specify programs. This will work in a minority of cases, but this restricted area is of importance.[2] In Sections 5 and 6 we consider two examples of program specification with CVN.

Exercises. Let $V_g = \mathbb{N}$ and let the relation $<$ (smaller than) be given.

6. Determine whether the following expressions are well-defined:

   (a) $T \mathrel{\cdot\!\!\rightarrow} 6$

   (b) $3 \mathrel{\cdot\!\!\rightarrow} 5$

   (c) $T \mathrel{\cdot\!\!\rightarrow} T$

   (d) $(3 \mathrel{{}_{\circ}{=}} 5) \mathrel{\cdot\!\!\rightarrow} 2 \nvrightarrow D$

7. Determine the *type* (data type, data domain) of the following expressions:

   (a) $T \mathrel{\cdot\!\!\rightarrow} 6$

   (b) $F \mathrel{\cdot\!\!\rightarrow} D$

   (c) $3 \mathrel{{}_{\circ}{=}} 5$

   (d) $(3 \mathrel{{}_{\circ}{=}} 5) \mathrel{\cdot\!\!\rightarrow} T$

   (e) $(3 \mathrel{{}_{\circ}{=}} 5) \mathrel{\cdot\!\!\rightarrow} 2 \nvrightarrow 7$

8. Argue whether the following holds:

$$\begin{aligned} X \propto Y \quad = \quad & g(X) \mathbin{{}_{\circ}\!\wedge} g(Y) \mathrel{\cdot\!\!\rightarrow} (X < Y) \nvrightarrow \\ & g(X) \mathbin{{}_{\circ}\!\wedge} \neg g(Y) \mathrel{\cdot\!\!\rightarrow} Y \nvrightarrow \\ & \neg g(X) \mathbin{{}_{\circ}\!\wedge} g(Y) \mathrel{\cdot\!\!\rightarrow} X \nvrightarrow \\ & \neg g(X) \mathbin{{}_{\circ}\!\wedge} \neg g(Y) \mathrel{\cdot\!\!\rightarrow} X. \end{aligned}$$

   Can you find a shorter CVN expression that defines $\propto$?

[2]Of course, many natural extensions of CVN are conceivable (e.g., with recursive or higher order ingredients). Some proposals are discussed later in the paper.

Motivation for CVN. BCMP strongly depends on the availability of a functional specification of a piece of software beforehand. The task of the BCMP project is to obtain an implementation of that specification.

Clearly such specifications must be expressed in a precise notation. There is a very rich literature about specification notations for software. Notoriously lacking, however, are simple techniques that can deal with less elegant aspects such as errors, divergencies and partiality of operators. Algebraic specification use a non-monotonic logic, but can in practice be based on monotonic term rewriting (see [6]), pseudo-algebraic specifications (present for instance in ASF + SDF [11]) use non-monotonic term rewriting instead. The logic of non-monotonic term rewriting is far from trivial unfortunately. Industrial strength methods like VDM use a three valued logic (see [16, 14]).

The notation CVN, as introduced in the present paper, can be used for the description of simple program functionalities. At the same time it deals with two kinds of errors: divergence (D) and abortion (M). CVN has been stripped from all aspects needed to make it applicable for more complex examples. No form of modularity is supported at all. This reflects the authors opinion that error handling cannot be viewed as an 'add on feature', whereas modularity does constitute an 'add on feature'. In particular the addition of error mechanisms to a complex specification notation may introduce intractable semantic questions. CVN addresses such questions right from the start. The price paid being that there is some 'overkill' in view of simple examples. Having secured a solution for error handling the development of more expressive formalisms can proceed in a systematic and uninterrupted way on top of CVN (when needed).

Two extensions of CVN. We end this section by introducing two 'natural' extensions of CVN. Let $\Phi : \mathbb{N} \to \mathbb{T}_4$ be a predicate on $\mathbb{N}$ (taking errors into account). The minimalisation operator μ ("mu") applied to Φ selects the (smallest) natural number n such that $\Phi(n) = T$ and $\forall k < n(\Phi(k) \in \{F, M\})$. If no such n exists, the minimalisation of Φ yields value D. In the latter case, minimalisation characterises either an infinite computation (for all $n \in \mathbb{N}$, $\Phi(n) \in \{F, M\}$), or one that breaks down ($\Phi(n) = D$ for certain n). As an example, let the predicate *odd* represent the odd numbers, then $\mu(odd) = 1$. A particular example is the case that $\Phi = \lambda n.M$, i.e. Φ is the predicate that yields value M for all $n \in \mathbb{N}$: now we find $\mu\Phi = D$. Taking S the *successor function* (i.e., $S(X) = X + 1$), minimalisation of Φ can be formally defined as follows:

$$\mu\Phi = \Phi(0) \cdot\!\!\to 0 \nrightarrow \mu(\lambda n.\Phi(S(n))).$$

Below we provide an example in which the minimalisation operator μ is used.

As a second extension of CVN, we introduce repeated conjunctions and disjunctions. For predicate $\Phi : \mathbb{N} \to \mathbb{T}_4$, we define:

$$
\begin{array}{ll}
{}_{\circ\circ}\!\!\bigwedge_i^k \Phi = T & \text{if } i > k \\[2ex]
{}_{\circ\circ}\!\!\bigwedge_i^k \Phi = \Phi(i) \mathbin{{}_{\circ}\!\wedge} {}_{\circ\circ}\!\!\bigwedge_{i+1}^k \Phi & \text{if } i \leq k
\end{array}
$$

and by dualization:

$$
\begin{array}{ll}
{}^{\circ\circ}\!\!\bigvee_i^k \Phi = F & \text{if } i > k \\[2ex]
{}^{\circ\circ}\!\!\bigvee_i^k \Phi = \Phi(l) \mathbin{{}^{\circ}\!\vee} {}^{\circ\circ}\!\!\bigvee_{i+1}^k \Phi & \text{if } i \leq k
\end{array}
$$

In the following example, both minimalisation and repeated conjunction occur.

Example. In the following we describe a prototypical, generic occurrence of D caused by partiality. Consider the partial subtraction function *psub* as introduced in Exercise 2, and recall our notation $p \cdot\!\!\to X \nrightarrow \neg p \cdot\!\!\to Y$ for **if** p **then** X **else** Y. So *psub* is defined by

$$
psub(X, Y) = (X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to 0 \nrightarrow \neg(X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to S(psub(X, S(Y))).
$$

In the case that at least one of X, Y is in $\{D, M\}$, this yields the expected value of $psub(X, Y)$. In the other case, so for $X, Y \in \mathbb{N}$, we give a further analysis of the definition of *psub*. We define $S^0(X) = X$ and $S^{k+1}(X) = S(S^k(X))$, and we further write $S(0) = 1$, $S^2(0) = 2$, etc. Unfolding the recursive definition of *psub* once, we obtain

$$
\begin{array}{rcl}
psub(X, Y) & = & (X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to 0 \nrightarrow \\
& & \neg(X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to [X \mathrel{{}_{\circ}\!\!=} S(Y) \cdot\!\!\to 1 \nrightarrow S^2(psub(X, S^2(Y)))] \\
& = & (X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to 0 \nrightarrow \\
& & \neg(X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to [X \mathrel{{}_{\circ}\!\!=} S(Y) \cdot\!\!\to 1] \nrightarrow \\
& & \neg(X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to [\neg(X \mathrel{{}_{\circ}\!\!=} S(Y)) \cdot\!\!\to S^2(psub(X, S^2(Y)))] \\
& = & (X \mathrel{{}_{\circ}\!\!=} Y) \cdot\!\!\to 0 \nrightarrow \\
& & [\neg(X \mathrel{{}_{\circ}\!\!=} Y) \mathbin{{}_{\circ}\!\wedge} X \mathrel{{}_{\circ}\!\!=} S(Y)] \cdot\!\!\to 1 \nrightarrow \\
& & [\neg(X \mathrel{{}_{\circ}\!\!=} Y) \mathbin{{}_{\circ}\!\wedge} \neg(X \mathrel{{}_{\circ}\!\!=} S(Y))] \cdot\!\!\to S^2(psub(X, S^2(Y))).
\end{array}
$$

Let $\phi_i^k(X, Y) = {}_{\circ\circ}\!\!\bigwedge_i^k \neg(X \mathrel{{}_{\circ}\!\!=} S^{i-1}(Y))$ and $\psi_i^k(X, Y) = \phi_i^k(X, Y) \mathbin{{}_{\circ}\!\wedge} X \mathrel{{}_{\circ}\!\!=} S^k(Y)$.

Extending the unfolding above, we can infer for any $k > 1$ that

$$\begin{aligned} psub(X,Y) \;&=\; \psi_1^0(X,Y) \cdot\!\!\rightarrow 0 \nrightarrow \psi_1^1(X,Y) \cdot\!\!\rightarrow 1 \nrightarrow ... \nrightarrow \psi_1^k(X,Y) \cdot\!\!\rightarrow k \nrightarrow \\ &\quad\; \psi_1^{k+1}(X,Y) \cdot\!\!\rightarrow S^{k+1}(psub(X, S^{k+1}(Y))). \end{aligned}$$

It is not hard to see that $\psi_1^k(X,Y) = X \mathrel{{}_{\sigma}{=}} S^k(Y)$ for X, Y ranging over $\mathbb{N}$, so

$$psub(X,Y) = (X \mathrel{{}_{\sigma}{=}} Y) \cdot\!\!\rightarrow 0 \nrightarrow (X \mathrel{{}_{\sigma}{=}} S(Y)) \cdot\!\!\rightarrow 1 \nrightarrow (X \mathrel{{}_{\sigma}{=}} S^2(Y)) \cdot\!\!\rightarrow 2 \nrightarrow ...$$

Now set $f = \lambda n. X \mathrel{{}_{\sigma}{=}} S^n(Y)$, then it follows that

$$psub(X,Y) = \mu f.$$

As an example, $psub(1,0) = \mu f(1,0) = 1$. Furthermore, by the fact that $0 \mathrel{{}_{\sigma}{=}} S^{k+1}(0) = F$ for all $k \in \mathbb{N}$, we obtain

$$psub(0,1) = D,$$

which reflects the infinite computation resulting from the recursive definition of $psub$.

Note that μf can be taken as the *definition* of $psub$ on $\mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{D\}$, and indeed reflects the partiality of $psub(X,Y)$ by a divergent computation when $Y > X$. Furthermore, μf also provides a correct definition for the extension of $psub$ to $\mathbb{N} \cup \{D\} \times \mathbb{N} \cup \{D\} \to \mathbb{N} \cup \{D\}$. However, on $\mathbb{N} \cup \{D, M\}$, $psub$ and μf define different operations: e.g., $psub(M,0) = M$ and $\mu f(M,0) = D$. This ends our example on the extensions of CVN.

Exercise.

8. Let $f = \mu.k \exists l, m, n((k+1)^{n+3} + (l+1)^{n+3} = (m+1)^{n+3} \nrightarrow 3)$. Argue—in a fashion similar to the argument given in the example—that $\mu f = D$. (Hint: Consult `http://www.netspace.org/herald/issues/092995/wiles.f.html`.)

5 Example A

In this section we consider the first of two examples, example **A**, of program specifications using CVN. In this and the second example we will use the notion of an *array*, i.e., a collection of indexed data. We also provide the description of a program (or better, a function) for the example and a black box tester both written in the programming language C designed by Dennis Ritchie at Bell Laboratories in about 1972 (see e.g. [15], [8]).

Input:

$max \in \mathbb{N}$,

integer array $a[0], ..., a[max]$.

We view $\{0, 1, 2, ..., max\}$ as V_g, and use notation

$[0, max]$

for this set. More precisely, $[i, j]$ represents the empty set if $i, j \in \{0, 1, 2, ..., max\}$ and $i > j$, and $\{i\} \cup [i + 1, j]$ otherwise.

Output in CVN:

$(max < 50) \cdot\!\!\rightarrow 2 \nrightarrow$

$(\forall i \in [25, max]\ (a[49] \geq a[i])) \cdot\!\!\rightarrow (a[1] + a[2]) \nrightarrow$

$(\forall i \in [0, max]\ (\forall j \in [0, max]\ (a[i] - a[j] \neq 83))) \cdot\!\!\rightarrow 12 \nrightarrow 0$

We remark that this functionality in itself is not an interesting one; it might occur in a larger collection of functionalities.

Table 1 exhibits a straightforward translation of the CVN specification into C code. The tester in Table 2 generates in `main` an array $a[0], a[1], \ldots, a[max]$ of random integers which — with the exception of $a[49]$ — are all within the interval [-`bound`,`bound`]. The length of the array, i.e., the number of integers generated, the upper and lower bound, and — in case `max` ≥ 50 — $a[49]$ are determined by the user. The produced array is written to the file `arrayA.txt`; the result computed by the function `ExampleA` is send to the standard output stream. Note that, since C integers are not arbitrary large, `bound` and $a[49]$ have to be in the range provided by the C compiler.

An exhaustive test set with `BlackboxA.c` will now contain a session similar to the one listed in Table 3. The first test yields clearly the expected result: since the generated array consists of 11 elements, the function returns 2. Inspection of the file `arrayA.txt` is in this case superfluous. Also the second test shows the right performance: the array has more than 50 elements and, since $a[49]$ coincides with the upper bound, it is an upper bound to the elements $a[25], \ldots, a[max]$. However, now we have to consult `arrayA.txt` in order to check that $-677 = a[1] + a[2]$. In the last test we generate a sufficiently long array with elements — with the exception of $a[49]$ — within the range $[-10, 10]$. $a[49]$ is assigned the value -11. $a[49]$ is therefore not an upper bound of the elements $a[25], \ldots, a[max]$, and the difference between the various elements does not exceed 21. The expected functionality is thus the return of the value 12, which is indeed the case.

```
typedef enum {false, true} bool;

int ExampleA ( int , int * );

int ExampleA ( int max , int *a )
{
  int i, j, result;
  bool cond;

  if ( max < 50 )
     result = 2;
  else {
      cond = true;
      i=25;
      while ( cond && i <= max ){
         if ( a[i] <= a[49] )
            i++;
         else
            cond = false;
      }
      if ( cond )
          result = a[1] + a[2];
      else {
          i = 0;
          cond =  true;
          while ( cond && i <= max ){
              j = 0;
              while ( cond && j <= max ){
                  if ( a[i] - a[j] != 83 )
                     j++;
                  else
                     cond = false;
              }
              i++;
          }
          if ( cond )
             result = 12;
          else
             result = 0;
      }
  }
  return result;
}
```

Table 1: `ExampleA.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ExampleA.c"

int main ( void )
{
  int i, b, max, bound, *a;
  FILE *output;

  printf( "Enter max: " );
  scanf( "%d", &max );
  printf( "Enter bound: " );
  scanf( "%d", &bound );
  srand( time(NULL) );
  output = fopen( "arrayA.txt", "w" );
  a = malloc(sizeof(i));
  for ( i=0; i<= max; i++ ){
    if ( i==49 ){
      printf( "Enter a[49]: " );
      scanf( "%d", a );
    } else {
      b = rand() % 2;
      if ( b == 0 ){
        *a = rand() % (bound + 1);
      } else {
        *a = - (rand() % (bound + 1));
      }
    }
    fprintf( output, "a[%d]=%6d\n" , i, *a );
    a++;
  }
  fclose( output );
  a = a - ( max + 1 );
  printf( "Result: %d\n", ExampleA ( max, a ) );
  return 0;
}
```

Table 2: `BlackboxA.c`

```
(bmcp@model 74) gcc -Wall -o BlackboxA BlackboxA.c
(bmcp@model 75) ./BlackboxA
Enter max: 10
Enter bound: 374
Result: 2
(bmcp@model 76) ./BlackboxA
Enter max: 460
Enter bound: 2000
Enter a[49]: 2000
Result: -677
(bmcp@model 77) more arrayA.txt
a[0]=  -927
a[1]=   928
a[2]= -1605
a[3]=   240
a[4]=  1809
.
.
.
.
(bmcp@model 78) ./BlackboxA
Enter max: 50
Enter bound: 10
Enter a[49]: -11
Result: 12
(bmcp@model 79)
```

Table 3: Typical test session for `ExampleA.c`

Exercise.

9. The test set depicted in Table 3 is not yet exhaustive, i.e., not all possible cases are taken into consideration. State the remaining case and design an appropriate test.

6 Example B

This section deals with the second example, Example B. We proceed along the same lines as in the previous section.

Input:

$$max \in \mathbb{N},$$
$$\text{integer array } a[0], ..., a[max],$$
$$\text{integer array } b[0], ..., b[max].$$

Again we set $V_g = \{0, 1, 2, ..., max\}$.

Output: We are interested in a program that will read $a[0], b[0]$, then produce $c[0]$, thereafter read $a[1], b[1]$ and produce $c[1]$ and so on till $c[max]$ has been generated. CVN is used to express $c[i]$ in $a[j], b[j]$ for $j \leq i$. The intuition is that $b[i]$ represents an instruction:

$$\begin{aligned} b[i] = 0 & \quad \text{represents insert,} \\ b[i] = 1 & \quad \text{represents delete,} \\ 0 \neq b[i] \neq 1 & \quad \text{represents test,} \end{aligned}$$

and $a[i]$ contains the value to be inserted, deleted or for which membership is tested. Initially the set which the program will maintain is empty. In CVN:

$$\begin{aligned} &((b[i] = 0) \mathbin{{}^{\circ}\!\vee} (b[i] = 1)) \cdot\!\!\rightarrow T \nrightarrow \\ &(i = 0) \cdot\!\!\rightarrow F \nrightarrow \\ &(\exists j\ 0 \leq j < i\ ((a[i] = a[j]) \mathbin{{}_{\circ}\!\wedge} (b[j] = 0) \mathbin{{}_{\circ}\!\wedge} \\ &\qquad \forall k\ j < k < i\ ((b[k] = 1) \mathbin{\circ\!\!\rightarrow} (a[k] \neq a[i])))) \cdot\!\!\rightarrow T \nrightarrow F \end{aligned}$$

Remark: again this functionality in itself is not an interesting one; it might occur in a larger collection of functionalities.

Table 4 contains the C code for Example B. The tester in Table 5 generates now in `main` two arrays $a[0], a[1], \ldots, a[max]$ and $b[0], b[1], \ldots, b[max]$ of random integers which are all within the interval [-`bounda`,`bounda`] and [0,`boundb`], respectively. The length of the arrays, the upper and lower bounds, and the index `i` of an element in the array $c[0], c[1], \ldots, c[max]$ are determined by the user. The produced arrays are written to the file `arrayB.txt`; the value of the element c[`i`] is send to the standard output stream. In Table 6 we listed a few example tests some of which require inspection of the generated arrays.

Exercise.

10. Design a complete series of tests for `ExampleB.c`

```
typedef enum {false, true} bool;
typedef struct {
  int a;
  int b;
} AB;

char ExampleB ( int , AB * );

char ExampleB ( int i , AB *ab )
{
  int j, k;
  bool cond;
  char result;

  if ( ab[i].b == 0 || ab[i].b == 1 ) {
     result = 'T';
  } else {
    if ( i == 0 ){
      result = 'F';
    } else {
      j = 0;
      cond = false;
      while ( !cond && j < i ){
        if ( ab[i].a == ab[j].a && ab[j].b == 0 ){
          k = j+1;
          cond = true;
          while ( cond && k < i ){
            if ( ab[k].b != 1 || ab[k].a != ab[i].a ){
              k++;
            } else cond = false;
          }
        }
        j++;
      }
      if ( cond ) result = 'T';
      else result = 'F';
    }
  }
  return result;
}
```

Table 4: `ExampleB.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "ExampleB.c"

int main ( void )
{
  int i, n, max, bounda, boundb;
  AB  *ab;
  FILE *output;

  printf( "Enter max:" );
  scanf( "%d", &max );
  printf( "Enter bounda:" );
  scanf( "%d", &bounda );
  printf( "Enter boundb:" );
  scanf( "%d", &boundb );
  srand( time(NULL) );
  output = fopen( "arrayB.txt", "w" );
  ab = malloc(sizeof(AB));
  for ( i = 0; i <= max; i++ ){
    n = rand() % 2;
    if ( n == 0 ){
      (*ab).a = rand() % (bounda + 1);
    } else {
      (*ab).a = - rand() % (bounda + 1);
    }
    (*ab).b = rand() % (boundb + 1);
    fprintf( output, "a[%d]=%6d \t b[%d]=%6d\n", i, (*ab).a, i, (*ab).b );
    ab++;
  }
  fclose( output );
  ab = ab - (max + 1);
  printf( "Enter i: " );
  scanf( "%d", &i );
  printf( "c[%d]=%c\n", i, ExampleB(i,ab) );
  return 0;
}
```

Table 5: BlackboxB.c

```
(bmcp@model 14) gcc -Wall -o BlackboxB BlackboxB.c
(bmcp@model 15) ./BlackboxB
Enter max:23
Enter bounda:41
Enter boundb:2
Enter i: 17
c[17]=T
(bmcp@model 16) ./BlackboxB
Enter max:34
Enter bounda:957
Enter boundb:1102
Enter i: 0
c[0]=F
(bmcp@model 17) more arrayB.txt
a[0]=   401      b[0]=   745
a[1]=   946      b[1]=    23
a[2]=   664      b[2]=   387
a[3]=  -892      b[3]=   478
a[4]=  -431      b[4]=    25
.                .
.                .
.                .
(bmcp@model 18) ./BlackboxB
Enter max:181
Enter bounda:749
Enter boundb:1275
Enter i: 1
c[1]=F
(bmcp@model 19) more arrayB.txt
a[0]=  -387   b[0]=   251
a[1]=   -48   b[1]=  1252
a[2]=  -141   b[2]=   480
a[3]=  -129   b[3]=   961
a[4]=  -470   b[4]=   464
.                .
.                .
.                .
(bmcp@model 20)
```

Table 6: Sample tests for ExampleB.c

References

[1] T. Abdel-Hamid and S.E. Madnick. *Software Project Dynamics. An Integrated Approach*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[2] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs* (2nd edition). Springer-Verlag, 1997.

[3] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[4] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Non-Classical Logics*, 5:199–217, 1995.

[5] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[6] J.A. Bergstra and J.V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the ACM*, 42(6):1194–1230, 1996.

[7] B. Boehm. *Software Engineering Economics*, Prentice Hall, Upper Saddle River, NJ, 1981.

[8] H.M. Deitel and P.J. Deitel. *C: How to Program* (second edition), Prentice Hall, 1999.

[9] T. DeMarco. *Controlling Software Projects. Management, Measurement and Estimation*, Yourdon Press, Prentice Hall, Englewood Cliffs, NY, 1982.

[10] T. DeMarco and T. Lister. *Peopleware. Productive Projects and Teams*, Dorset House Publishing Co., NY, 1982.

[11] A. van Deursen, J. Heering, and P. Klint (editors). *Language Prototyping: An Algebraic Specification Approach.* AMAST Series in Computing (Volume 5), World Scientific Publishing Co., 1996.

[12] D.P. Freedman and G.M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews* (3rd edition). Dorset House, 1990. (Originally published by Little, Brown & Company, 1982.)

[13] R.L. Glass. *Building Quality Software*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[14] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31(5):399–430, 1994.

[15] B.W. Kernighan and D.M. Ritchie. *The C Programming Language* (Second edition), Englewood Cliffs, NJ: Prentice Hall, 1988.

[16] C.A. Middelburg. VVSL: a language for structured VDM specifications. *Formal Aspects of Computing*, 1(1):115–135, 1989.

[17] G. Myers. *The Art of Software Testing.* Wiley, 1979.

[18] G.M. Weinberg. *Quality Software Management*, Dorset House Publishing, NY, 1992.

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.wins.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P0001] I. Bethke J.A. Bergstra and A. Ponse. *Basic Multi-Competence Programming.*

[P9906] J.A. Bergstra and M.E. Loots. *Software mechanics for Java multi-threading.*

[P9905] M.P.A. Sellink and C. Verhoef. *Towards Automated Modification of Legacy Assets.*

[P9904] M.P.A. Sellink and C. Verhoef. *Scaffolding for Software Renovation.*

[P9903] René Krikhaar, André Postma, M.P.A. Sellink, Marc Stroucken, and C. Verhoef. *A Two-phase Process for Software Architecture Improvement.*

[P9902] A. van Deursen, P. Klint, and C. Verhoef. *Research Issues in the Renovation of Legacy Systems.*

[P9901] M.P.A. Sellink and C. Verhoef. *Generation of Software Renovation Factories from Compilers.*

[P9811] J.A. Bergstra and M.E. Loots. *Program Algebra for Component Code.*

[P9810] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. *Restructuring of* COBOL/CICS *Legacy Systems.*

[P9809] B. Wedemeijer. *Introduction & Basic Tooling for* CASL *using* ASF+SDF.

[P9808] J.A. Bergstra and A. Ponse. *Two Recursive Generalizations of Iteration in Process Algebra.*

[P9807] M.P.A. Sellink and C. Verhoef. *An Architecture for Software Maintenance.*

[P9806] M. B. van der Zwaag. *Some Verifications in Process Algebra with Iota.*

[P9805] M.P.A. Sellink and C. Verhoef. *Development, Assessment, and Reengineering of Language Descriptions.*

[P9804] M.P.A. Sellink and C. Verhoef. *Native Patterns.*

[P9803] J.J. Brunekreef. *Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language* Alma-0.

[P9802] W. Fokkink and C. Verhoef. *Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories.*

[P9801] P. Klint and C. Verhoef. *Evolutionary Software Engineering: A Component-based Approach.*

[P9726] J.A. Bergstra and A. Ponse. *Grid Protocol Specifications.*

[P9725] J.A. Bergstra and A. Ponse. *Process Algebra Primitives for File Transfer.*

[P9724] J.A. Bergstra and A. Ponse. *Process Algebra with Four-Valued Logic.*

[P9723] J.A. Bergstra and A. Ponse. *Kleene's Three-Valued Logic and Process Algebra.*

[P9722] J.A. Bergstra and A. Ponse. *Bochvar-McCarthy Logic and Process Algebra.*

[P9721] M.P.A. Sellink and C. Verhoef. *Reflections on the Evolution of COBOL.*

[P9720] M. van der Graaf. *A Specification of Box to HTML in ASF+SDF.*

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF*'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for μCRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*

[P9420] M.G.J. van den Brand and E. Visser. *From Box to TeX: An algebraic approach to the construction of documentation tools.*

[P9419] J.C.M. Baeten, J.A. Bergstra, and Gh. Stefanescu. *Process algebra with feedback.*

[P9418] L.H. Oei. *Pruning the search tree of interlocking design and application language operational semantics.*

[P9417] B. Diertens. *New features in PSF I: interrupts, disrupts, and priorities.*

[P9416] S.M. Üsküdarlı. *Generating visual editors for formally specified languages.*

[P9415] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. *Control and data transfer in the distributed editor of the ASF+SDF meta-environment.*

[P9414] M.G.J. van den Brand and C. Groza. *The algebraic specification of annotated abstract syntax trees.*

[P9413] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen (editors). *Workshop on Algebra of Communicating Processes May 16-17, 1994 Utrecht University.*

[P9218b] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing probabilistic processes: ACP with generative probabilities (revised version of P9218).*

[P9412] C. Groza. *An experiment in implementing process algebra specifications in a procedural language.*

[P9411] M.J. Koens and L.H. Oei. *A real time muCRL specification of a system for traffic regulation at signalized intersections.*

[P9410] J.J. van Wamel. *Inductive proofs with sets, and some applications in process algebra.*

[P9409] J.F. Groote and J.J. van Wamel. *Algebraic data types and induction in muCRL.*

[P9408] J.A. Bergstra and P. Klint. *The toolbus: a component interconnection architecture.*

[P9407] P.H. Rodenburg. *Module algebra for initial algebra semantics.*

[P9406] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. *A propositional logic with 4 values: true, false, divergent and meaningless.*

[P9211b] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in real time process algebra (revised version of P9211).*

[P9405] J.C.M. Baeten and J.A. Bergstra. *Process algebra with partial choice.*

[P9314b] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with iteration and nesting (revised version of P9314).*

[P9404] J. van den Brink and W.O.D. Griffioen. *Formal semantics of discrete absolute timed interworkings.*

[P9403] J.A. Bergstra and Gh. Stefanescu. *Processes with multiple entries and exits modulo isomorphism and modulo bisimulation.*

[P9402] J.A. Bergstra and Ch. Stefanescu. *Bisimulation is two-way simulation.*

[P9401] J.C.M. Baeten and J.A. Bergstra. *Graph isomorphism models for non interleaving process algebra.*

[P9331] J.A. Hillebrand. *An algebraic specification of a manufacturing system with hierarchical control.*

[P9330] K.M.M. de Leeuw and H. van der Meer. *A turning grille from the ancestral castle of the Dutch Stadtholders.*

[P9329] J.J. Brunekreef. *Process specification in a UNITY format.*

[P9328] B. Diertens. *A simulator for PSF in PSF.*

[P9327] M.G.J. van den Brand. *Generation of language independent prettyprinters.*

[P9326] J.C.M. Baeten and J.A. Bergstra. *Non interleaving process algebra.*

[P9306b] J.A. Bergstra, A. Ponse, and J.J. van Wamel. *Process algebra with backtracking (revised version of P9306).*

[P9325] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra with infinitesimals.*

[P9324] J.J. Brunekreef, R.L.C. Koymans J.P. Katoen, and S. Mauw. *Design and analysis of dynamic leader election protocols in broadcast networks.*

[P9323] M. Kaart and I. Polak. *Het alternating bit protocol met time out in discrete tijd.*

[P9322] N.J. Drost. *Unification in an algebra with choice and sequential composition.*

[P9321] N.J. Drost. *Unification in an algebra with choice and action prefix.*

[P9320] K.M.M. de Leeuw and H. van der Meer. *Een roostergeheimschrift van Alexander baron van Spaen.*

[P9319] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with combinators.*

[P9318] I. Bethke and A. Ponse. *A car registration authority, a concise PSF-specification.*

[P9317] S.F.M. van Vlijmen and J.J. van Wamel. *A semantic approach to Protocold using process algebra.*

[P9316] J.J. Brunekreef and A. Ponse. *An algebraic specification of a model factory, part IV.*

[P9315] M.G.J. van den Brand. *Prettyprinting without losing comments.*

[P9314] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with iteration.*

[P9313] S.F.M. van Vlijmen and A. van Waveren. *Algebraic specification of a system for traffic regulation at signalized intersections.*

[P9312] J.W.C. Koorn and H.C.N. Bakker. *Building an editor from existing components: an exercise in software re-use.*

[P9311] I. Polak. *Specification of a bank account in PSF.*

[P9305b] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix (revised version of P9305).*

[P9310] A.S. Klusener, S.F.M. van Vlijmen, and A. van Waveren. *Service independent building blocks-I; concepts, examples and formal specifications.*

[P9309] J.C.M. Baeten, J.A. Bergstra, and R.N. Bol. *A real time process logic.*

[P9308] J.F.A.K. van Benthem and J.A. Bergstra. *Logic of transition systems.*

[P9307] A. Stins and A. Schoneveld. *Specification of a bank account with process algebra.*

[P9306] J.A. Bergstra, A. Ponse, and J.J. van Wamel. *Process algebra with backtracking.*

[P9305] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix.*

[P9304] S.F.M. van Vlijmen and A. van Waveren. *On generating synchronous interworkings from PSF process traces.*

[P9303] A. Ponse and J.A. Verschuren. *An algebraic specification of a model factory, part III.*

[P9302] J.A. Bergstra and J.V. Tucker. *The data type variety of stack algebras.*

[P9301] J.J. van Wamel. *A library for PSF.*

[P9208b] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208).*

[P9222] J.W.C. Koorn. *Connecting semantic tools to a syntax-directed user-interface.*

[P9221] J. Blanco. *Definability with the state operator in process algebra.*

[P9220] A. van Waveren. *Specification of remote sensing mechanisms in real space process algebra.*

[P9219] I. Bethke and P.H. Rodenburg. *Equational Constructor Induction.*

[P9218] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing Probabilistic Processes: ACP with Generative Probabilities.*

[P9217] A. Ponse. *Process algebra and dynamic logic.*

[P9005b] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra (revised version).*

[P9216] E.E. Polak. *An efficient implementation of branching bisimulation and distinguishing formulae.*

[P9215] J.A. Bergstra and J.V. Tucker. *Equational specifications, complete term rewriting systems, and computable and semicomputable algebras.*

[P9214] J. A. Hillebrand and A. Ponse. *An algebraic specification of a model factory, part II.*

[P9213] N.J. Drost. *Unification in the algebra of sets with union and empty set.*

[P9212] J.J. van Wamel. *A study of a one bit sliding window protocol in ACP.*

[P9211] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in realtime process algebra.*

[P9210] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. *Decidability of bisimulation equivalence for processes generating context-free languages.*

[P9209] S.F.M. van Vlijmen and A. van Waveren. *An algebraic specification of a model factory.*

[P9208] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra.*

[P9207] K. de Leeuw and H. van der Meer. *A secret message in the archive of the last great pensionary of Holland.*

[P9206] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9205] J.J. van Wamel. *An algebraic verification of the concurrent alternating bit protocol.*

[P9204] C. Verhoef. *On induction principles.*

[P9203] J.A. Verschuren. *A simulator for mCRL in ASF+SDF.*

[P9202] J.W.C. Koorn. *A generic text and structure editor.*

[P9201] P.H. Rodenburg. *Interpolation in Equational Logic.*

[P9102b] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols (revised version of P9102).*

[P9112] A. Mateescu. *Recursive systems with auxiliary variables in Basic Process Algebra.*

[P9111] J.C.M. Baeten and J.A. Bergstra. *A survey of axiom systems for process algebras.*

[P9110] J.J. van Wamel. *A formal specification of three simple protocols.*

[P9109] J.J. Brunekreef. *A formal specification of two simple protocols for Local Area Networks.*

[P9108] J.J. Brunekreef. *A formal specification of the Amoeba Transaction Protocol.*

[P9107] G.J. Veltink. *The PSF toolkit.*

[P9106] J.C.M. Baeten and J.A. Bergstra. *Asynchronous communication in real space process algebra.*

[P9105] C. Verhoef. *An operator definition principle (for process algebras).*

[P9104] J.C.M. Baeten and J.A. Bergstra. *The state operator in real time process algebra.*

[P9103] H.P. Korver. *Computing distinguishing formulas for branching bisimulation.*

[P9102] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols.*

[P9101] H.P. Korver. *The current state of bisimulation tools.*

[P9009] G.J. Veltink. *From PSF to TIL.*

[P9008] J.C.M. Baeten and J.A. Bergstra. *Process algebra with signals and conditions.*

[P9007] S. Mauw and Gy. Max. *A formal specification of the Ethernet protocol.*

[P9006] G.J. Akkerman and J.C.M. Baeten. *Term rewriting analysis in process algebra.*

[P9005] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9004] N.J. Drost. *Algbraic formulations of trace theory.*

[P9002b] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero a object (revised version of P9002).*

[P8804b] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions (revised version of P8804).*

[P8910b] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (revised version of P8910).*

[P8906b] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD (revised version of P8906).*

[P9003] C. Verhoef. *On the register operator.*

[P8916b] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra (revised version of P8916).*

[P9002] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero object and non-determinacy.*

[P9001] L.W. Kuurman. *The jungle of process semantics.*

[P8916] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra.*

[P8915] J.C. Mulder. *The inevitable coffee machine.*

[P8914] H. Jacobsson and S. Mauw. *A Token ring network in PSFd.*

[P8913] H.R. Walters. *Hybrid implementations of algebraic specifications.*

[P8912] S. Mauw and G.J. Veltink. *A Tool Interface Language for PSF.*

[P8911] P.H. Rodenburg. *Interpolation in conditional equational logic.*

[P8910] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (preliminary version).*

[P8909] J.A. Bergstra. *Algebra of states and transitions.*

[P8908] S. Mauw and F. Wiedijk. *Specification of the transit node in PSFd.*

[P8808b] J.A. Bergstra. *A mode transfer operator in process algebra (revised version of P8808).*

[P8907] J.A. Bergstra. *Kerninformatica en toekomst.*

[P8906] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD.*

[P8815b] J.A. Bergstra. *Process algebra for synchronous communication and observation (revised version of P8815).*

[P8905] J.A. Bergstra and J.W. Klop. *BMACP.*

[P8904] J.A. Bergstra, S. Mauw, and F. Wiedijk. *Uniform algebraic specifications of finite sorts with equality.*

[P8903] J.A. Bergstra. *A representation of addition and deletion lists using module algebra.*

[P8902] R.P. Ogilvie. *Knowledge engineering vs. software engineering.*

[P8901] S. Mauw and G.J. Veltink. *An introduction to PSF.*

[P8826] J. Treur. *Design of modular and interactive knowledge-based systems.*

[P8825] H.W. Lenferink. *Local control of inference by means of meta-rules.*

[P8824] J.A. Bergstra and J.W. Klop. *Process theory based on bisimulation semantics.*

[P8823] J.A. Bergstra, J. Heering, and P. Klint. *Module Algebra (revised version).*

[P8822] J.L.M. Vrancken. *The algebraic specification of semicomputable data types (revised version).*

[P8821] J.C.M. Baeten and F.W. Vaandrager. *Specification and verification of a circuit in ACP (revised version).*

[P8820] J. Treur. *A logical framework for design processes.*

[P8819] H.K. Faber. *Strategische uitleg in een medisch expertsysteem.*

[P8818] H.K. Faber. *Uitleg en kennisrepresentatie in expertsystemen.*

[P8817] J. Treur. *Heuristic reasoning with incomplete knowledge.*

[P8816] J.A. Bergstra. *ACP with signals.*

[P8815] J.A. Bergstra. *Process algebra for synchronous communication and observation.*

[P8814] S. Mauw and G.J. Veltink. *A process specification formalism.*

[P8813] J. Treur. *Reasoning about partial models, actions and plans.*

[P8812] J. Treur. *On the use of reflection principles in modelling complex reasoning.*

[P8811] F. Wiedijk. *Voorlopig rapport over de specificatie-taal Perspect.*

[P8810] R.E. Swart. *The BCA Bull Course Adviser.*

[P8809] J. Treur. *Metakennis en meta-inferenties in expertsystemen.*

[P8808] J.A. Bergstra. *A mode transfer operator in process algebra.*

[P8807] J.L.M. Vrancken. *The implementation of process algebra specifications in POOL-T.*

[P8806] J. Treur. *Generic inference processes and their interactions in complex diagnostic tasks; a logical description.*

[P8805] J. Treur. *Completeness and definability in diagnostic expert systems.*

[P8804] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions.*

[P8803] J.C.M. Baeten and F.W. Vaandrager. *Specification and verification of a circuit in ACP.*

[P8802] J.C.M. Baeten and J.A. Bergstra. *Recursive process definitions with the state operator.*

[P8801] F.R. Burggraaff and E. van der Meulen. *ASF Specification of a B-tree of order 1.*

[P8714] L.G. Bouma and H.R. Walters. *Implementing algebraic specifications.*

[P8713] W. Syski. *On a certain probabilistic approximation method for reasoning with uncertainty.*

[P8712] W. Syski and J. Treur. *Reasoning about uncertainty represented by meta-reasoning, the endorsements approach.*

[P8711] J. Treur. *Een logische analyse van diagnostische redeneerprocessen; redeneren met en over hypothesen.*

[P8710] J. Treur. *Volledigheid en definieerbaarheid in diagnostische redeneersystemen.*

[P8709] J.C.M. Baeten and J.A. Bergstra. *Global renaming operators in concrete process algebra (revised version).*

[P8708] S. Mauw. *Process algebra as a tool for the specification and verification of CIM-architectures.*

[P8707] J.H. Verhagen. *Expertsystemen bij de Nederlandse Spoorwegen.*

[P8706] J.C.M. Baeten, J.A. Bergstra, and J.L.M. Vrancken. *Processen en procesexpressies.*

[P8705] J.L.M. Vrancken. *The algebraic specification of semicomputable datatypes.*

[P8704] F. Wiedijk. *Termherschrijfsystemen in Prolog.*

[P8703] A.V. Hurkmans. *Een declaratieve en procedurele kennisrepresentatievorm voor kennissystemen, toegepast op NEXT.*

[P8702] M.R. Dasselaar. *Development of an expert system, from theory to practice.*

[P8701] R.A. Groenveld. *Verification of a sliding window protocol by means of process algebra.*