**University of Amsterdam**
Programming Research Group

# Thread Extraction for Polyadic Instruction Sequences

J.A. Bergstra
C.A. Middelburg

Report PRG0803 February 2008

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel. +31 20 525.7591
e-mail: janb@science.uva.nl

C.A. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

# Thread Extraction for Polyadic Instruction Sequences*

J.A. Bergstra[1,2] and C.A. Middelburg[1]

[1] Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
[2] Department of Philosophy, Utrecht University,
P.O. Box 80126, 3508 TC Utrecht, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

**Abstract.** Instruction sequences are often fragmented. An important reason for instruction sequence fragmentation is that the execution architecture at hand to execute instruction sequences sets bounds to the size of instruction sequences. In this paper, we study instruction sequences that have been split into fragments. The purpose is to develop a theoretical understanding of this matter. The possible joint behaviours exhibited by a collection of fragments on execution are explained in terms of threads as considered in basic thread algebra. In this way, a setting is provided in which the slow-down results of instruction sequence fragmentation can be analysed.

*Keywords:* instruction sequence fragmentation, polyadic instruction sequence, thread extraction, basic thread algebra.

*1998 ACM Computing Classification:* D.3.1, D.3.3, F.1.1, F.3.2, F.3.3.

## 1 Introduction

In this paper, we consider fragmentation of sequential programs that take the form of instruction sequences. With that we carry on the line of research with which a start was made in [3]. This line of research concerns the development of a theoretical understanding of possible forms of sequential programs and associated ways of programming. It is a fact of life that instruction sequences are often fragmented. An important reason for instruction sequence fragmentation is that the execution architecture at hand to execute instruction sequences sets bounds to the size of instruction sequences. However, there may also be other reasons for instruction sequence fragmentation, for instance business economical reasons. In this paper, we look for a direct approach to the formalization of instruction sequences that have been split into fragments and we investigate their relationship with sequential programs.

The question is how a joint behaviour of the fragments in a collection of fragments is achieved. The view of this matter is that there can only be a single

* This research was partly carried out in the framework of the Jacquard-project Symbiosis, which is funded by the Netherlands Organisation for Scientific Research (NWO).

fragment being executed at any stage, but the fragment in question may make any fragment in the collection the one being executed by means of a special instruction for switching over execution to another fragment. Each fragment can be taken for the one being executed initially. This means that there is generally not a unique joint behaviour. Therefore, the fragments are not considered to constitute a single sequential program. As the view is taken in the line of research carried on in this paper that the behaviour exhibited by a sequential program on execution is a thread as considered in basic thread algebra [3],[3] so the view is taken here that the possible joint behaviours exhibited by the fragments on execution are threads as considered in basic thread algebra. In execution architectures, a fragment must be loaded in order to become the one being executed. Hence, making a fragment the one being executed can be looked upon as loading that fragment for execution.

The instruction sequences taken for fragments are called polyadic instruction sequences in this paper. We introduce polyadic instruction sequences in the setting of program algebra [3], an algebra of programs in which programs are looked upon as instruction sequences. In [3], a hierarchy of program notations rooted in program algebra is presented as well. Included in this hierarchy are very simple program notations which are close to existing assembly languages up to and including simple program notations that support structured programming by offering a rendering of conditional and loop constructs. All of these program notations are referred to in this paper, but only one of them is actually used. That program notation is introduced under the name PGLD in [3].

This paper is organized as follows. First, we review basic thread algebra, program algebra, and the program notation PGLD (Sections 2, 3, and 4). Next, we introduce polyadic instruction sequences in the setting of program algebra, explain the possible joint behaviours of a collection of polyadic instruction sequences using basic thread algebra, and give an example of the use of polyadic instruction sequences (Sections 5 and 6). Following this, we extend basic thread algebra with a mechanism for interaction between threads and services, introduce a state-based approach to describe services, and give a state-based description of instruction register file services (Sections 7, 8, and 9). After that, we show that, for each possible joint behaviour of a collection of polyadic instruction sequences, a sequential program can be synthesized from the collection of polyadic instruction sequences that exhibits on execution essentially the behaviour in question by interaction with an instruction register file service (Section 10). Finally, we make some concluding remarks (Section 11).

## 2 Basic Thread Algebra

In this section, we review BTA, a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under

[3] In [3], basic thread algebra is introduced under the name basic polarized process algebra. Prompted by the development of thread algebra [8], which is a design on top of it, basic polarized process algebra has been renamed to basic thread algebra.

**Table 1.** Axiom of BTA

| | |
|---|---|
| $x \trianglelefteq \iota \trianglerighteq y = x \trianglelefteq \iota \trianglerighteq x$ | T1 |

execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that there are fixed but arbitrary finite sets $\mathcal{A}$ and $\mathcal{I}$ with $\mathcal{A} \cap \mathcal{I} = \emptyset$ and $\mathsf{tau} \in \mathcal{I}$. The members of $\mathcal{A}$ are called *basic actions* and the members of $\mathcal{I}$ are called *internal actions*. The members of $\mathcal{A} \cup \mathcal{I}$ are referred to as *actions*.

The intuition is that each basic action performed by a thread is taken as a command to be processed by a service provided by the execution environment of the thread. The processing of a command may involve a change of state of the service concerned. At completion of the processing of the command, the service produces a reply value. This reply is either $\mathsf{T}$ or $\mathsf{F}$ and is returned to the thread concerned. Performing an internal action will never lead to a state change and always lead to the reply $\mathsf{T}$, but notwithstanding all that it is a concrete action: its presence matters.

In previous work, we take in essence the singleton set $\{\mathsf{tau}\}$ for $\mathcal{I}$. The generalization made here permits internal actions with differences relevant for analysis to be distinguished.

Although BTA is one-sorted, we make this sort explicit. The reason for this is that we will extend BTA with an additional sort in Section 7.

The algebraic theory BTA has one sort: the sort $\mathbf{T}$ of *threads*. To build terms of sort $\mathbf{T}$, BTA has the following constants and operators:

- the *deadlock* constant $\mathsf{D} : \mathbf{T}$;
- the *termination* constant $\mathsf{S} : \mathbf{T}$;
- for each $a \in \mathcal{A} \cup \mathcal{I}$, the binary *postconditional composition* operator $\_ \trianglelefteq a \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

Terms of sort $\mathbf{T}$ are built as usual (see e.g. [11, 12]). Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{T}$, including $x, y, z$.

We use infix notation for postconditional composition. We introduce *action prefixing* as an abbreviation: $a \circ p$, where $p$ is a term of sort $\mathbf{T}$, abbreviates $p \trianglelefteq a \trianglerighteq p$.

Let $p$ and $q$ be closed terms of sort $\mathbf{T}$ and $a \in \mathcal{A} \cup \mathcal{I}$. Then $p \trianglelefteq a \trianglerighteq q$ will perform action $a$, and after that proceed as $p$ if the processing of $a$ leads to the reply $\mathsf{T}$ (called a positive reply), and proceed as $q$ if the processing of $a$ leads to the reply $\mathsf{F}$ (called a negative reply).

BTA has only one axiom. This axiom is given in Table 1. In this table, $\iota$ stands for an arbitrary member of $\mathcal{I}$. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \iota \trianglerighteq y = \iota \circ x$.

Each closed BTA term of sort $\mathbf{T}$ denotes a finite thread, i.e. a thread of which the length of the sequences of actions that it can perform is bounded. Guarded recursive specifications give rise to infinite threads.

**Table 2.** Axioms for guarded recursion

| | | |
|---|---|---|
| $\langle X\|E\rangle = \langle t_X\|E\rangle$ | if $X = t_X \in E$ | RDP |
| $E \Rightarrow X = \langle X\|E\rangle$ | if $X \in \mathrm{V}(E)$ | RSP |

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where $V$ is a set of variables of sort $\mathbf{T}$ and each $t_X$ is a term of the form $\mathsf{D}$, $\mathsf{S}$ or $t \trianglelefteq a \trianglerighteq t'$ with $t$ and $t'$ BTA terms of sort $\mathbf{T}$ that contain only variables from $V$. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in $E$. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [1]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

We extend BTA with guarded recursion by adding constants for solutions of guarded recursive specifications and axioms concerning these additional constants. For each guarded recursive specification $E$ and each $X \in \mathrm{V}(E)$, we add a constant of sort $\mathbf{T}$ standing for the unique solution of $E$ for $X$ to the constants of BTA. The constant standing for the unique solution of $E$ for $X$ is denoted by $\langle X|E\rangle$. Moreover, we add the axioms for guarded recursion given in Table 2 to BTA, where we write $\langle t_X|E\rangle$ for $t_X$ with, for all $Y \in \mathrm{V}(E)$, all occurrences of $Y$ in $t_X$ replaced by $\langle Y|E\rangle$. In this table, $X$, $t_X$ and $E$ stand for an arbitrary variable of sort $\mathbf{T}$, an arbitrary BTA term of sort $\mathbf{T}$ and an arbitrary guarded recursive specification over BTA, respectively. Side conditions are added to restrict the variables, terms and guarded recursive specifications for which $X$, $t_X$ and $E$ stand. The equations $\langle X|E\rangle = \langle t_X|E\rangle$ for a fixed $E$ express that the constants $\langle X|E\rangle$ make up a solution of $E$. The conditional equations $E \Rightarrow X = \langle X|E\rangle$ express that this solution is the only one.

We will use the following abbreviation: $a^\omega$, where $a \in \mathcal{A} \cup \mathcal{I}$, abbreviates $\langle X|\{X = a \circ X\}\rangle$.

We will write BTA+REC for BTA extended with the constants for solutions of guarded recursive specifications and axioms RDP and RSP.

In [5], we show that the threads considered in BTA+REC can be viewed as processes that are definable over ACP [10].

Closed terms of sort $\mathbf{T}$ from the language of BTA+REC that denote the same infinite thread cannot always be proved equal by means of the axioms of BTA+REC. We introduce the approximation induction principle to remedy this. The approximation induction principle, AIP in short, is based on the view that two threads are identical if their approximations up to any finite depth are identical. The approximation up to depth $n$ of a thread is obtained by cutting it off after performing a sequence of actions of length $n$.

AIP is the infinitary conditional equation given in Table 3. Here, following [3], approximation of depth $n$ is phrased in terms of a unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. The axioms for the projection operators are given in Table 4. In this table, $a$ stands for an arbitrary member of $\mathcal{A} \cup \mathcal{I}$.

**Table 3.** Approximation induction principle

| |
|---|
| $\bigwedge_{n\geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$ AIP |

**Table 4.** Axioms for projection operators

| | |
|---|---|
| $\pi_0(x) = \mathsf{D}$ | P0 |
| $\pi_{n+1}(\mathsf{S}) = \mathsf{S}$ | P1 |
| $\pi_{n+1}(\mathsf{D}) = \mathsf{D}$ | P2 |
| $\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$ | P3 |

We will write BTA+REC+AIP for BTA+REC extended with the projection operators and the axioms from Tables 3 and 4.

## 3 Program Algebra

In this section, we review PGA, an algebra of sequential programs based on the idea that sequential programs are in essence sequences of instructions. PGA provides a program notation for finite-state threads.

In PGA, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}$ of *basic instructions*. PGA has the following *primitive instructions*:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* $a$;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- a *termination instruction* !.

We write $\mathfrak{I}$ for the set of all primitive instructions.

The intuition is that the execution of a basic instruction $a$ may modify a state and produces $\mathsf{T}$ or $\mathsf{F}$ at its completion. In the case of a positive test instruction $+a$, basic instruction $a$ is executed and execution proceeds with the next primitive instruction if $\mathsf{T}$ is produced and otherwise the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. In the case where $\mathsf{T}$ is produced and there is not at least one subsequent primitive instruction and in the case where $\mathsf{F}$ is produced and there are not at least two subsequent primitive instructions, deadlock occurs. In the case of a negative test instruction $-a$, the role of the value produced is reversed. In the case of a plain basic instruction $a$, the value produced is disregarded: execution always proceeds as if $\mathsf{T}$ is produced. The effect of a forward jump instruction $\#l$ is that execution proceeds with the $l$-th next instruction of the program concerned. If $l$ equals 0 or the $l$-th next instruction does not exist, then $\#l$ results in deadlock. The effect of the termination instruction ! is that execution terminates.

PGA has the following constants and operators:

**Table 5.** Axioms of PGA

| | |
|---|---|
| $(x\,;y)\,;z = x\,;(y\,;z)$ | PGA1 |
| $(x^n)^\omega = x^\omega$ | PGA2 |
| $x^\omega\,;y = x^\omega$ | PGA3 |
| $(x\,;y)^\omega = x\,;(y\,;x)^\omega$ | PGA4 |

**Table 6.** Defining equations for thread extraction operation of PGA

| | |
|---|---|
| $\lvert a\rvert = a \circ \mathsf{D}$ | $\lvert\#l\rvert = \mathsf{D}$ |
| $\lvert a\,;x\rvert = a \circ \lvert x\rvert$ | $\lvert\#0\,;x\rvert = \mathsf{D}$ |
| $\lvert{+}a\rvert = a \circ \mathsf{D}$ | $\lvert\#1\,;x\rvert = \lvert x\rvert$ |
| $\lvert{+}a\,;x\rvert = \lvert x\rvert \trianglelefteq a \trianglerighteq \lvert\#2\,;x\rvert$ | $\lvert\#l+2\,;u\rvert = \mathsf{D}$ |
| $\lvert{-}a\rvert = a \circ \mathsf{D}$ | $\lvert\#l+2\,;u\,;x\rvert = \lvert\#l+1\,;x\rvert$ |
| $\lvert{-}a\,;x\rvert = \lvert\#2\,;x\rvert \trianglelefteq a \trianglerighteq \lvert x\rvert$ | $\lvert!\rvert = \mathsf{S}$ |
| | $\lvert!\,;x\rvert = \mathsf{S}$ |

- for each $u \in \mathfrak{I}$, an *instruction* constant $u$;
- the binary *concatenation* operator $\_\,;\,\_$;
- the unary *repetition* operator $\_^\omega$.

Terms are built as usual. Throughout the paper, we assume that there are infinitely many variables, including $x, y, z$.

We use infix notation for concatenation and postfix notation for repetition.

Closed PGA terms are considered to denote programs. The intuition is that a program is in essence a non-empty, finite or periodic infinite sequence of primitive instructions.[4] These sequences are called *single pass instruction sequences* because PGA has been designed to enable single pass execution of instruction sequences: each instruction can be dropped after it has been executed. Programs are considered equal if they represent the same single pass instruction sequence. The axioms for instruction sequence equivalence are given in Table 5. In this table, $n$ stands for an arbitrary natural number greater than 0. For each $n > 0$, the term $x^n$ is defined by induction on $n$ as follows: $x^1 = x$ and $x^{n+1} = x\,;x^n$. The *unfolding* equation $x^\omega = x\,;x^\omega$ is derivable. Each closed PGA term is derivably equal to a term in *canonical form*, i.e. a term of the form $P$ or $P\,;Q^\omega$, where $P$ and $Q$ are closed PGA terms that do not contain the repetition operator.

Each closed PGA term is considered to denote a program of which the behaviour is a finite-state thread, taking the set $\mathfrak{A}$ of basic instructions for the set $\mathcal{A}$ of basic actions. The *thread extraction* operation $\lvert\_\rvert$ assigns a thread to each program. The thread extraction operation is defined by the equations given in Table 6 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$ and $u \in \mathfrak{I}$) and the rule given in Table 7. This rule is ex-

[4] A periodic infinite sequence is an infinite sequence with only finitely many subsequences.

**Table 7.** Rule for cyclic jump chains

| |
|---|
| $x \cong \#0\,;y \Rightarrow \lvert x\rvert = \mathsf{D}$ |

**Table 8.** Defining formulas for structural congruence predicate

| |
|---|
| $\#n+1\,;u_1\,;\ldots;u_n\,;\#0 \cong \#0\,;u_1\,;\ldots;u_n\,;\#0$ |
| $\#n+1\,;u_1\,;\ldots;u_n\,;\#m \cong \#m+n+1\,;u_1\,;\ldots;u_n\,;\#m$ |
| $(\#n+l+1\,;u_1\,;\ldots;u_n)^\omega \cong (\#l\,;u_1\,;\ldots;u_n)^\omega$ |
| $\#m+n+l+2\,;u_1\,;\ldots;u_n\,;(v_1\,;\ldots;v_{m+1})^\omega \cong$ |
| $\#n+l+1\,;u_1\,;\ldots;u_n\,;(v_1\,;\ldots;v_{m+1})^\omega$ |
| $x \cong x$ |
| $x_1 \cong y_1 \wedge x_2 \cong y_2 \Rightarrow x_1\,;x_2 \cong y_1\,;y_2 \wedge {x_1}^\omega \cong {y_1}^\omega$ |

pressed in terms of the *structural congruence* predicate $\_ \cong \_$, which is defined by the formulas given in Table 8 (for $n, m, l \in \mathbb{N}$ and $u_1, \ldots, u_n, v_1, \ldots, v_{m+1} \in \mathfrak{I}$).

The equations given in Table 6 do not cover the case where there is a cyclic chain of forward jumps. Programs are structural congruent if they are the same after removing all chains of forward jumps in favour of single jumps. Because a cyclic chain of forward jumps corresponds to #0, the rule from Table 7 can be read as follows: if $x$ starts with a cyclic chain of forward jumps, then $|x|$ equals $\mathsf{D}$. It is easy to see that the thread extraction operation assigns the same thread to structurally congruent programs. Therefore, the rule from Table 7 can be replaced by the following generalization: $x \cong y \Rightarrow |x| = |y|$.

The behaviour of each closed PGA term, is a thread that is definable by a finite guarded recursive specification over BTA. The other way round, each finite guarded recursive specification over BTA in which no internal actions occur can be translated into a closed PGA term of which the behaviour is the solution of the finite guarded recursive specification concerned.

Closed PGA terms are considered to denote programs and therefore they constitute an elementary program notation. Closed PGA terms are also called PGA programs.

In [3], a hierarchy of program notations rooted in PGA is presented. In this hierarchy, the program notations PGLA, PGLB, PGLC, PGLD, PGLDg, PGLE, and PGLS appear. PGLA programs are translated into PGA programs by means of a function `pgla2pga`, PGLB programs are translated into PGLA programs by means of a function `pglb2pgla`, PGLC programs are translated into PGLB programs by means of a function `pglc2pglb`, etcetera. These functions are called projections in [3]. Each of them translates each of the programs from a program notation higher in the hierarchy into a program from a program notation lower in the hierarchy that exhibits the same behaviour on execution. Moreover, PGA programs are translated into PGLA programs by means of a function `pga2pgla`, PGLA programs are translated into PGLB programs by means of a function `pgla2pglb`, PGLB programs are translated into PGLC programs by means of a

function $\mathtt{pglb2pglc}$, etcetera. These functions are called embeddings in [3]. Each of them translates each of the programs from a program notation lower in the hierarchy into a program from a program notation higher in the hierarchy that exhibits the same behaviour on execution. We remark that there does not exist a function by which each PGLE program is translated into a PGLS program that exhibits the same behaviour on execution: PGLS is strictly weaker than PGLE.

In Section 5, we will use the function $\mathtt{pgla2pga}$ and in addition the functions $\mathtt{pglb2pga}$, $\mathtt{pglc2pga}$, ... defined by $\mathtt{pglb2pga}(P) = \mathtt{pgla2pga}(\mathtt{pglb2pgla}(P))$ for all PGLB programs $P$, $\mathtt{pglc2pga}(P) = \mathtt{pglb2pga}(\mathtt{pglc2pglb}(P))$ for all PGLC programs $P$, etcetera. In Section 10, we will use the function $\mathtt{pglc2pgld}$ and in and in addition the functions $\mathtt{pga2pglc}$ defined by $\mathtt{pga2pglc}(P) = \mathtt{pglb2pglc}(\mathtt{pgla2pglb}(\mathtt{pga2pgla}(P)))$ for all PGA programs $P$.

PGLC, PGLD and PGLS are the most interesting program notations of the ones mentioned above. PGLC and PGLD are close to existing assembly languages. The main difference between them is that PGLC has relative jump instructions and PGLD has absolute jump instructions. PGLS supports structured programming by offering a rendering of conditional and loop constructs instead of (unstructured) jump instructions.

## 4 The Program Notation PGLD

In this section, we review the program notation PGLD. This program notation will be used later on in Sections 6 and 10.

In PGLD, like in PGA, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}$ of *basic instructions*. Again, the intuition is that the execution of a basic instruction $a$ may modify a state and produces $\mathsf{T}$ or $\mathsf{F}$ at its completion.

PGLD has the following primitive instructions:

- for each $a \in \mathfrak{A}$, a *plain basic instruction* $a$;
- for each $a \in \mathfrak{A}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, an *absolute jump instruction* $\#\#l$.

PGLD programs have the form $u_1; \ldots ; u_k$, where $u_1, \ldots, u_k$ are primitive instructions of PGLD.

The plain basic instructions, the positive test instructions, and the negative test instructions are as in PGA. The effect of an absolute jump instruction $\#\#l$ is that execution proceeds with the $l$-th instruction of the program concerned. If $\#\#l$ is itself the $l$-th instruction, then deadlock occurs. If $l$ equals 0 or $l$ is greater than the length of the program, then termination occurs.

The function $\mathtt{pgld2pga}$ from the set of all PGLD programs to the set of all PGA programs can be defined directly as follows:

$$\mathtt{pgld2pga}(u_1 \,;\, \ldots \,;\, u_k) = (\psi_1(u_1) \,;\, \ldots \,;\, \psi_k(u_k) \,;\, ! \,;\, ! \,;\, \#0 \,;\, \#0)^\omega \;,$$

where the auxiliary functions $\psi_j$ from the set of all primitive instructions of PGLD to the set of all primitive instructions of PGA are defined as follows ($1 \leq j \leq k$):

$$
\begin{aligned}
&\psi_j(\#\#l) = \#l - j && \text{if } j \leq l \leq k\ ,\\
&\psi_j(\#\#l) = \#k + 2 - (j - l) && \text{if } 0 < l < j\ ,\\
&\psi_j(\#\#l) = ! && \text{if } l = 0 \vee l > k\ ,\\
&\psi_j(u) \quad = u && \text{if } u \text{ is not a jump instruction}\ .
\end{aligned}
$$

## 5 Polyadic Instruction Sequences

In this section, we look for a direct approach to the formalization of instruction sequences that have been split into fragments. The fragments resulting from a splitting are considered instruction sequences with special instructions for switching over execution from one fragment to another. The instruction sequences concerned are called polyadic instruction sequences.

It is assumed that a special version of PGLA, PGLB, PGLC, PGLD, PGLDg, PGLE or PGLS is used for each polyadic instruction sequence. Moreover, it is assumed that a collection of polyadic instruction sequences between which execution can be switched takes the form of a sequence, called a polyadic instruction sequence vector, in which each polyadic instruction sequence is coupled with the program notation used for it.

Our general view on the way of achieving a joint behaviour of the polyadic instruction sequences in a polyadic instruction sequence vector is as follows:

- there can only be a single polyadic instruction sequence being executed at any stage;
- the polyadic instruction sequence in question may make any polyadic instruction sequence in the vector the one being executed;
- making another polyadic instruction sequence the one being executed is effected by executing a special instruction for switching over execution;
- any polyadic instruction sequence can be taken for the one being executed initially.

In addition to special instructions for switching over execution, polyadic instruction sequences may contain special instructions for putting instructions into instruction registers and special instructions which are actually instruction place-holders: each of them is replaced by the instruction contained in one of the instruction registers on making a polyadic instruction sequence the one being executed. The presence of special instructions of the latter kind turns a polyadic instruction sequence into a parameterized instruction sequence of which the parameters are filled in each time it is made the one being executed. This feature accounts for the use of the prefix polyadic. Its merit is primarily that it allows for execution to proceed in effect from different positions each time a polyadic instruction sequence is loaded for execution. An example of this is given in Section 6.

We take the line that different program notations can be used for different polyadic instruction sequences in a polyadic instruction sequence vector. On making a polyadic instruction sequence in the vector the one being executed, it is considered to be translated into a $\mathrm{PGA_p}$ program.

$\mathrm{PGA_p}$ is a variant of PGA in which the above-mentioned special instructions are incorporated. In $\mathrm{PGA_p}$, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}_\mathrm{c}$ of *core basic instructions*. In $\mathrm{PGA_p}$, a basic instruction is either a core basic instruction or a supplementary basic instructions.

$\mathrm{PGA_p}$ has the following *core primitive instructions*:

- for each $a \in \mathfrak{A}_\mathrm{c}$, a *plain basic instruction* $a$;
- for each $a \in \mathfrak{A}_\mathrm{c}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{A}_\mathrm{c}$, a *negative test instruction* $-a$;
- for each $l \in \mathbb{N}$, a *forward jump instruction* $\#l$;
- a *termination instruction* !.

We write $\mathfrak{I}_\mathrm{c}$ for the set of all core primitive instructions. The core primitive instructions of $\mathrm{PGA_p}$ are the pendants of the primitive instructions of PGA.

$\mathrm{PGA_p}$ has the following *supplementary basic instructions*:

- for each $i \in \mathbb{N}$, a *switch-over instruction* $\#\#\#i$;
- for each $i \in \mathbb{N}$ and $u \in \mathfrak{I}_\mathrm{c}$, a *put instruction* $\$\mathsf{put}{:}i{:}u$;
- for each $i \in \mathbb{N}$, a *get instruction* $\$\mathsf{get}{:}i$.

We write $\mathfrak{A}_\mathrm{s}$ for the set of all supplementary basic instructions. In the presence of a polyadic instruction sequence vector, a switch-over instruction $\#\#\#i$ is the instruction for switching over execution to the $i$-th polyadic instruction sequence in the vector. A put instruction $\$\mathsf{put}{:}i{:}u$ is the instruction for putting instruction $u$ in the instruction register with number $i$. A get instruction $\$\mathsf{get}{:}i$ is the instruction place-holder of which each occurrence in a polyadic instruction sequence is replaced by the contents of the instruction register with number $i$ on switching over execution to that polyadic instruction sequence.

The supplementary basic instructions of $\mathrm{PGA_p}$ can be viewed as built-in basic instructions. However, as laid down below, supplementary basic instructions do not occur in positive or negative test instructions. Thus, the core primitive instructions and supplementary basic instructions make up the primitive instructions of $\mathrm{PGA_p}$.

$\mathrm{PGA_p}$ has the following constants and operators:

- for each $u \in \mathfrak{I}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$, an *instruction* constant $u$ ;
- the binary *concatenation* operator $\_ \,;\, \_$ ;
- the unary *repetition* operator $\_^\omega$ .

The axioms of $\mathrm{PGA_p}$ are the same as the axioms of PGA.

Suppose that in PGA the restriction is dropped that $\mathfrak{A}$ must be a finite set. Then $\mathrm{PGA_p}$ can be viewed as the specialization of PGA obtained by taking the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ for $\mathfrak{A}$ and excluding terms in which basic instructions from $\mathfrak{A}_\mathrm{s}$ occur in positive or negative test instructions. Henceforth, we actually drop the restriction that $\mathfrak{A}$ must be a finite set. This simplifies the definitions of the different program notations that can be used for polyadic instruction sequences and also enables the use of the functions `pgla2pga`, `pglb2pga`, etcetera for translating programs in those program notations into $\mathrm{PGA_p}$ programs.

The different program notations that can be used for polyadic instruction sequences are $\mathrm{PGLA_p}$, $\mathrm{PGLB_p}$, $\mathrm{PGLC_p}$, $\mathrm{PGLD_p}$, $\mathrm{PGLDg_p}$, $\mathrm{PGLE_p}$, and $\mathrm{PGLS_p}$. The set of all $\mathrm{PGLA_p}$ programs is the subset of the set of all PGLA programs, taking the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ for $\mathfrak{A}$, in which the basic instructions from $\mathfrak{A}_\mathrm{s}$ do not occur in positive or negative test instructions. $\mathrm{PGLB_p}$, $\mathrm{PGLC_p}$, $\mathrm{PGLD_p}$, $\mathrm{PGLDg_p}$, $\mathrm{PGLE_p}$, and $\mathrm{PGLS_p}$ are defined similarly.

If the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ is taken for $\mathfrak{A}$, the function `pgla2pga` translates each $\mathrm{PGLA_p}$ program into a $\mathrm{PGA_p}$ program that exhibits the same behaviour on execution. Similar remarks apply to $\mathrm{PGLB_p}$, $\mathrm{PGLC_p}$, $\mathrm{PGLD_p}$, $\mathrm{PGLDg_p}$, $\mathrm{PGLE_p}$, and $\mathrm{PGLS_p}$.

A *polyadic instruction sequence* is either a $\mathrm{PGLA_p}$ program, a $\mathrm{PGLB_p}$ program, a $\mathrm{PGLC_p}$ program, a $\mathrm{PGLD_p}$ program, a $\mathrm{PGLDg_p}$ program, a $\mathrm{PGLE_p}$ program or a $\mathrm{PGLS_p}$ program.

A *polyadic instruction sequence vector* is a sequence of pairs consisting of a polyadic instruction sequence and a member of the set $\{A, B, C, D, Dg, E, S\}$ of *program notation indices*.

Let $\alpha$ be a polyadic instruction sequence vector, let $P_1, \ldots, P_n$ and $c_1, \ldots, c_n$ be polyadic instruction sequences and program notation indices, respectively, such that $\alpha = \langle (P_1, c_1) \rangle \frown \ldots \frown \langle (P_n, c_n) \rangle$,[5] and let $i \in [1, n]$. Then we write $pg(\alpha, i)$ and $pgn(\alpha, i)$ for $P_i$ and $c_i$, respectively.

Let $\alpha$ be a polyadic instruction sequence vector of length $n$, and let $i \in [1, n]$. Then program notation index $pgn(\alpha, i)$ indicates which program notation is used for polyadic instruction sequence $pg(\alpha, i)$. $A$ stands for $\mathrm{PGLA_p}$, $B$ stands for $\mathrm{PGLB_p}$, etcetera. The program notation used is made explicit because it cannot always be determined uniquely from the polyadic instruction sequence concerned, whereas the behaviour that this polyadic instruction sequence exhibits on execution may be different for each of the program notations in question.

The set of instruction registers that contain an instruction and the contents of each of those registers matter when a polyadic instruction sequence is made the one being executed. That makes us introduce the notion of an instruction register file state and special notation relating to this notion.

An *instruction register file state* is a function $\sigma : I \to \mathfrak{I}_\mathrm{c}$, where $I$ is a finite subset of $\mathbb{N}$.

Let $p$ be a $\mathrm{PGA_p}$ program and $\sigma$ be a instruction register file state. Then we write $p[\sigma]$ for $p$ with, for all $i \in \mathrm{dom}(\sigma)$, all occurrences of $\$\mathsf{get}{:}i$ in $p$ replaced by $\sigma(i)$.

Let $i, n \in \mathbb{N}$ be such that $1 \leq i \leq n$, let $\alpha$ be a polyadic instruction sequence vector of length $n$, and let $\sigma$ be a instruction register file state. Then we write $valid(\alpha, i, \sigma)$ to indicate that instructions of the form $\$\mathsf{get}{:}i$ do not occur in $prj_{pgn(\alpha,i)}(pg(\alpha, i))[\sigma]$.

[5] We write $D^*$ for the set of all finite sequences with elements from set $D$. We use the following notation for finite sequences: $\langle\,\rangle$ for the empty sequence, $\langle d \rangle$ for the sequence having $d$ as sole element, $\sigma \frown \sigma'$ for the concatenation of finite sequences $\sigma$ and $\sigma'$, and $\mathrm{len}(\sigma)$ for the length of finite sequence $\sigma$.

We also use special notation relating to the program notation indices. Let $c$ be a program notation index. Then we write $prj_c$ for the projection `pgla2pga` if $c = A$, the projection `pglb2pga` if $c = B$, etcetera.

An obvious choice of the thread extraction operation of $\mathrm{PGA_p}$ is the thread extraction operation of PGA, taking the set $\mathfrak{A}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ for $\mathfrak{A}$, restricted to the set of closed terms of $\mathrm{PGA_p}$. This thread extraction operation is considered not to be the proper one, because it treats the supplementary basic instructions as arbitrary basic instructions and thus disregards the fixed effects that they produce on execution. Moreover, this thread extraction operation requires that in BTA the restriction is dropped that $\mathcal{A}$ must be a finite set.

As regards the proper thread extraction for $\mathrm{PGA_p}$, the idea is that it yields, for each $\mathrm{PGA_p}$ program $P$, a function that gives, for each polyadic instruction sequence vector $\alpha$, the thread that is the joint behaviour of $P$ and the polyadic instruction sequences in $\alpha$ if $P$ is the polyadic instruction sequence being executed initially. Because this behaviour depends upon the set of instruction registers that contain an instruction and the contents of each of those registers, we need a thread extraction operation for each instruction register file state.

For each instruction register file state $\sigma$, we introduce the *thread extraction* operation $|\_|_\sigma$. These thread extraction operations are defined by the equations given in Table 9 (for $a \in \mathfrak{A}$, $l, i \in \mathbb{N}$, $u \in \mathfrak{I}_\mathrm{c} \cup \mathfrak{A}_\mathrm{s}$ and $v \in \mathfrak{I}_\mathrm{c}$). and the rule given in Table 10. Here, it is assumed that $\mathsf{gl} \in \mathcal{I}$. The internal action $\mathsf{gl}$ (generate and load) represents the internal activity involved in switching over execution. The internal actions $\mathsf{tau}$ and $\mathsf{gl}$ are distinguished because $\mathsf{tau}$ is considered to represent a negligible internal activity, whereas $\mathsf{gl}$ is considered to represent a substantial internal activity.

We can couple nominal indices as labels with some of the polyadic instruction sequences in a polyadic instruction sequence vector. This would permit the use of alternative switch-over instructions with nominal indices instead of ordinal indices, like with the goto instructions from PGLDg. In the notational style of [2], the form of those alternative switch-over instructions would be $\#\#\#[i]$.

## 6 Example

In this section, we consider the splitting of a PGLD program $P$ of 10000 instructions into two fragments.

We write $\nu_1(l)$ for the number of absolute jump instructions $\#\#l'$ with $l' > 5000$ from position 1 up to position $l$ and $\nu_2(l)$ for the number of absolute jump instructions $\#\#l'$ with $l' \leq 5000$ from position 5001 up to position $l$.

The polyadic instruction sequence $P'$ corresponding to the first half of $P$ is obtained from the first half of $P$ as follows:

- the instruction $\$\mathsf{get}{:}1$ is prefixed to it;
- each absolute jump instructions $\#\#l$ with $l \leq 5000$ is replaced by the absolute jump instructions $\#\#l'$, where $l' = l + \nu_1(l) + 1$;
- each absolute jump instructions $\#\#l$ with $l > 5000$ is replaced by the instruction sequence $\$\mathsf{put}{:}2{:}\#l' \,;\, \#\#\#2$, where $l' = (l - 5000) + \nu_2(l - 5000)$;

**Table 9.** Defining equations for thread extraction operations of $\mathrm{PGA_p}$

$$
\begin{array}{ll}
|a|_\sigma(\alpha) = a \circ \mathsf{D} & \\
|a\,;x|_\sigma(\alpha) = a \circ |x|_\sigma(\alpha) & \\
|{+}a|_\sigma(\alpha) = a \circ \mathsf{D} & \\
|{+}a\,;x|_\sigma(\alpha) = |x|_\sigma(\alpha) \trianglelefteq a \trianglerighteq |\#2\,;x|_\sigma(\alpha) & \\
|{-}a|_\sigma(\alpha) = a \circ \mathsf{D} & \\
|{-}a\,;x|_\sigma(\alpha) = |\#2\,;x|_\sigma(\alpha) \trianglelefteq a \trianglerighteq |x|_\sigma(\alpha) & \\
 & \\
|\#l|_\sigma(\alpha) = \mathsf{D} & \\
|\#0\,;x|_\sigma(\alpha) = \mathsf{D} & \\
|\#1\,;x|_\sigma(\alpha) = |x|_\sigma(\alpha) & \\
|\#l+2\,;u|_\sigma(\alpha) = \mathsf{D} & \\
|\#l+2\,;u\,;x|_\sigma(\alpha) = |\#l+1\,;x|_\sigma(\alpha) & \\
|!|_\sigma(\alpha) = \mathsf{S} & \\
|!\,;x|_\sigma(\alpha) = \mathsf{S} & \\
 & \\
|\#\#\#i|_\sigma(\alpha) = \mathsf{gl} \circ |prj_{pgn(\alpha,i)}(pg(\alpha,i))[\sigma]|_\sigma(\alpha) & \text{if } 1 \leq i \leq n \wedge valid(\alpha,i,\sigma) \\
|\#\#\#i|_\sigma(\alpha) = \mathsf{D} & \text{if } 1 \leq i \leq n \wedge \neg\, valid(\alpha,i,\sigma) \\
|\#\#\#i|_\sigma(\alpha) = \mathsf{S} & \text{if } i = 0 \vee i > n \\
|\#\#\#i\,;x|_\sigma(\alpha) = \mathsf{gl} \circ |prj_{pgn(\alpha,i)}(pg(\alpha,i))[\sigma]|_\sigma(\alpha) & \text{if } 1 \leq i \leq n \wedge valid(\alpha,i,\sigma) \\
|\#\#\#i\,;x|_\sigma(\alpha) = \mathsf{D} & \text{if } 1 \leq i \leq n \wedge \neg\, valid(\alpha,i,\sigma) \\
|\#\#\#i\,;x|_\sigma(\alpha) = \mathsf{S} & \text{if } i = 0 \vee i > n \\
|\$\mathsf{put}{:}i{:}v|_\sigma(\alpha) = \mathsf{tau} \circ \mathsf{D} & \\
|\$\mathsf{put}{:}i{:}v\,;x|_\sigma(\alpha) = \mathsf{tau} \circ |x|_{\sigma \oplus [i \mapsto u]}(\alpha) & \\
|\$\mathsf{get}{:}i|_\sigma(\alpha) = \mathsf{D} & \\
|\$\mathsf{get}{:}i\,;x|_\sigma(\alpha) = \mathsf{D} &
\end{array}
$$

**Table 10.** Rule for cyclic jump chains

$$\frac{}{x \cong \#0\,;y \;\Rightarrow\; |x|_\sigma(\alpha) = \mathsf{D}}$$

and the polyadic instruction sequence $P''$ corresponding to the second half of $P$ is obtained from the second half of $P$ as follows:

- the instruction $\$\mathsf{get}{:}2$ is prefixed to it;
- each absolute jump instructions $\#\#l$ with $l > 5000$ is replaced by the absolute jump instructions $\#\#l'$, where $l' = (l - 5000) + \nu_2(l - 5000) + 1$;
- each absolute jump instructions $\#\#l$ with $l \leq 5000$ is replaced by the instruction sequence $\$\mathsf{put}{:}1{:}\#l'\,;\#\#\#1$, where $l' = l + \nu_1(l)$.

Notice that the positions occurring in jump instructions are adapted to the prefixing of a get instruction to each half of $P$ and the replacement of each jump instructions that gives rise to a jump into the other half of $P$ by two instructions.

For any instruction register file state $\sigma$, we have that $|P|$ coincides with $|\mathsf{\$put{:}1{:}}\#1 \mathbin{;} \#\#\#1|_\sigma(\langle(P', D)\rangle \frown \langle(P'', D)\rangle)$ after the presence of the internal actions tau and gl in the latter behaviour has been concealed. In Section 7, we will introduce operators to conceal the presence of internal actions.

In this section, we have illustrated by means of an example that splitting a program into fragments is relatively simple. In Section 10, we will show that synthesizing a program from a collection of fragments is fairly complicated.

## 7 Interaction of Threads with Services

A thread may perform certain basic actions only for the sake of having itself affected by some service. When processing a basic action performed by a thread, a service affects that thread by returning a reply value to the thread at completion of the processing of the basic action. In this section, we introduce the use mechanism, which is concerned with this kind of interaction between threads and services.[6] In the current paper, we will use the use mechanism to have behaviours of PGA programs affected by some service. We also introduce an abstraction mechanism. This mechanism serves for concealment of the presence of internal actions, which arise among other things from the use mechanism.

It is assumed that there is a fixed but arbitrary finite set $\mathcal{F}$ of *foci* and a fixed but arbitrary finite set $\mathcal{M}$ of *methods*. Each focus plays the role of a name of a service provided by the execution environment that can be requested to process a command. Each method plays the role of a command proper. For the set $\mathcal{A}$ of basic actions, we take the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Performing a basic action $f.m$ is taken as making a request to the service named $f$ to process command $m$.

We introduce yet another sort: the sort $\mathbf{S}$ of *services*. However, we will not introduce constants and operators to build terms of this sort. $\mathbf{S}$ is considered to stand for the set of all services. We identify services with functions $H : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$ that satisfy the following condition:

$$\forall \alpha \in \mathcal{M}^+, m \in \mathcal{M} \bullet (H(\alpha) = \mathsf{B} \Rightarrow H(\alpha \frown \langle m \rangle) = \mathsf{B}) \;.$$

Given a service $H$ and a method $m \in \mathcal{M}$, the *derived service* of $H$ after processing $m$, written $\frac{\partial}{\partial m} H$, is defined by $\frac{\partial}{\partial m} H(\alpha) = H(\langle m \rangle \frown \alpha)$.

A service $H$ can be understood as follows:

- if $H(\langle m \rangle) = \mathsf{T}$, then the request to process $m$ is accepted by the service, the reply is positive, and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(\langle m \rangle) = \mathsf{F}$, then the request to process $m$ is accepted by the service, the reply is negative, and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(\langle m \rangle) = \mathsf{B}$, then the request to process $m$ is rejected by the service.

For each $f \in \mathcal{F}$, we introduce the binary *use* operator $\_ /_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. Intuitively, $p /_f H$ is the thread that results from processing all basic actions

[6] This version of the use mechanism was first introduced in [8]. In later papers, it is also called thread-service composition.

**Table 11.** Axioms for use operators

| | | |
|---|---|---|
| $\mathsf{S} \mathbin{/_f} H = \mathsf{S}$ | | TSC1 |
| $\mathsf{D} \mathbin{/_f} H = \mathsf{D}$ | | TSC2 |
| $\mathsf{tau} \circ x \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} H)$ | | TSC3 |
| $(x \trianglelefteq g.m \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq g.m \trianglerighteq (y \mathbin{/_f} H)$ | if $f \neq g$ | TSC4 |
| $(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} \frac{\partial}{\partial m} H)$ | if $H(\langle m \rangle) = \mathsf{T}$ | TSC5 |
| $(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ (y \mathbin{/_f} \frac{\partial}{\partial m} H)$ | if $H(\langle m \rangle) = \mathsf{F}$ | TSC6 |
| $(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{D}$ | if $H(\langle m \rangle) = \mathsf{B}$ | TSC7 |

**Table 12.** Axioms for abstraction operators

| | | |
|---|---|---|
| $\tau_\iota(\mathsf{S}) = \mathsf{S}$ | | TT1 |
| $\tau_\iota(\mathsf{D}) = \mathsf{D}$ | | TT2 |
| $\tau_\iota(\iota \circ x) = \tau_\iota(x)$ | | TT3 |
| $\tau_\iota(x \trianglelefteq a \trianglerighteq y) = \tau_\iota(x) \trianglelefteq a \trianglerighteq \tau_\iota(y)$ | if $a \neq \iota$ | TT4 |

performed by thread $p$ that are of the form $f.m$ by service $H$. When a basic action of the form $f.m$ performed by thread $p$ is processed by service $H$, it is turned into the internal action $\mathsf{tau}$ and postconditional composition is removed in favour of action prefixing on the basis of the reply value produced.

The axioms for the use operators are given in Table 11. In this table, $f$ and $g$ stand for an arbitrary foci from $\mathcal{F}$ and $m$ stands for an arbitrary method from $\mathcal{M}$. Axioms TSC3 and TSC4 express that internal actions and basic actions of the form $g.m$ with $f \neq g$ are not processed. Axioms TSC5 and TSC6 express that a thread is affected by a service as described above when a basic action of the form $f.m$ performed by the thread is processed by the service. Axiom TSC7 expresses that deadlock takes place when a basic action to be processed is not accepted.

Let $T$ stand for either BTA, BTA+REC or BTA+REC+AIP. Then we will write $T$+TSC for $T$, taking the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ for $\mathcal{A}$, extended with the use operators and the axioms from Table 11.

In [5], we show that the services considered here can be viewed as processes that are definable over an extension of ACP with conditionals introduced in [4].

Both basic actions and internal actions are actions whose presence matters. For each $\iota \in \mathcal{I}$, we introduce the unary *abstraction* operator $\tau_\iota : \mathbf{T} \to \mathbf{T}$ to conceal the presence of internal action $\iota$ in the case where its presence does not matter after all.

The axioms for the abstraction operators are given in Table 12. In this table, $a$ stands for an arbitrary action from $\mathcal{A} \cup \mathcal{I}$.

A main difference between the version of the use mechanism introduced here and the version of the use mechanism introduced in [9] is that the former version does not incorporate abstraction and the latter version incorporates abstraction.

Let $T$ stand for either BTA, BTA+REC, BTA+REC+AIP, BTA+TSC, BTA+REC+TSC or BTA+REC+AIP+TSC. Then we will write $T$+ABSTR for $T$ extended with the abstraction operators and the axioms from Table 12.

For each $\iota \in \mathcal{I}$, the equation $\tau_\iota(\iota^\omega) = \mathsf{D}$ is derivable from the axioms of BTA+REC+AIP+ABSTR.

## 8 State-Based Description of Services

In this section, we introduce the state-based approach to describe families of services that will be used in Section 9. This approach is similar to the approach to describe state machines introduced in [9].

In this approach, a family of services is described by

- a set of states $S$;
- an effect function $\mathit{eff} : \mathcal{M} \times S \to S$;
- a yield function $\mathit{yld} : \mathcal{M} \times S \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$;

satisfying the following conditions:

$$\begin{aligned}&\exists s \in S \bullet \forall m \in \mathcal{M} \bullet \\ &\quad (\mathit{yld}(m, s) = \mathsf{B} \wedge \forall s' \in S \bullet (\mathit{yld}(m, s') = \mathsf{B} \Rightarrow \mathit{eff}(m, s') = s)) \,.\end{aligned}$$

The set $S$ contains the states in which the services may be, and the functions *eff* and *yld* give, for each method $m$ and state $s$, the state and reply, respectively, that result from processing $m$ in state $s$.

We define, for each $s \in S$, a cumulative effect function $\mathit{ceff}_s : \mathcal{M}^* \to S$ in terms of $s$ and *eff* as follows:

$$\begin{aligned}&\mathit{ceff}_s(\langle\rangle) = s \,, \\ &\mathit{ceff}_s(\alpha \frown \langle m \rangle) = \mathit{eff}(m, \mathit{ceff}_s(\alpha)) \,.\end{aligned}$$

We define, for each $s \in S$, a service $H_s$ in terms of $\mathit{ceff}_s$ and *yld* as follows:

$$H_s = \mathit{yld}(m, \mathit{ceff}_s(\alpha)) \,.$$

$H_s$ is called the service with *initial state* $s$ described by $S$, *eff* and *yld*. We say that $\{H_s \mid s \in S\}$ is the *family of services* described by $S$, *eff* and *yld*.

The condition that is imposed on $S$, *eff* and *yld* implies that, for each $s \in S$, $H_s$ is a service indeed. It is worth mentioning that $\frac{\partial}{\partial m} H_s = H_{\mathit{eff}(m,s)}$.

## 9 Instruction Register File Services

In this section, we give a state-based description of a simple family of services that constitute a register file with a finite set of registers that can contain instructions from a finite set of core primitive instructions. These services will be used in Section 10.

It is assumed that a fixed but arbitrary set $I \subseteq \mathbb{N}$ such that $I = [1, n]$ for some $n \in \mathbb{N}$ and a fixed but arbitrary finite set $U \subseteq \mathfrak{I}_{\mathrm{c}}$ have been given. The set $I$ is considered to contain the positions of the registers in the instruction register file and the set $U$ is considered to contain the instructions that can be put in those registers.

We write $\mathit{IRFS}$ for the set $\bigcup_{I' \subseteq I} I' \to U$. The members of $\mathit{IRFS}$ are considered to be the possible instruction register file states. It is assumed that a fixed but arbitrary bijection $\theta : \mathit{IRFS} \to [1, \mathrm{card}(\mathit{IRFS})]$ has been given.

The instruction register file services accept the following methods:

- for each $i \in I$ and $u \in U$, a *register put method* $\mathsf{put}{:}i{:}u$;
- for each $j \in \mathrm{rng}(\theta)$, a *register file test method* $\mathsf{eq}{:}j$.

We write $\mathcal{M}_{\mathsf{irf}}$ for the set $\{\mathsf{put}{:}i{:}u \mid i \in I \wedge u \in U\} \cup \{\mathsf{eq}{:}j \mid j \in \mathrm{rng}(\theta)\}$.

It is assumed that $\mathcal{M}_{\mathsf{irf}} \subseteq \mathcal{M}$.

The methods accepted by instruction register file services can be explained as follows:

- $\mathsf{put}{:}i{:}u$ : the contents of register $i$ becomes instruction $u$ and the reply is $\mathsf{T}$;
- $\mathsf{eq}{:}j$ : if the state of the instruction register file equals $\theta^{-1}(j)$, then nothing changes and the reply is $\mathsf{T}$; otherwise nothing changes and the reply is $\mathsf{F}$.

Let $s \in \mathit{IRFS}$. Then we write $\mathcal{IRF}_s$ for the service with initial state $s$ described by $S = \mathit{IRFS} \cup \{\uparrow\}$, where $\uparrow \notin \mathit{IRFS}$, and the functions *eff* and *yld* defined as follows ($\sigma \in \mathit{IRFS}$):[7]

$$
\begin{array}{lll}
\mathit{eff}(\mathsf{put}{:}i{:}u, \sigma) = \sigma \oplus [i \mapsto u] \,, & & \mathit{yld}(\mathsf{put}{:}i{:}n, \sigma) = \mathsf{T} \,, \\
\mathit{eff}(\mathsf{eq}{:}j, \sigma) = \sigma \,, & & \mathit{yld}(\mathsf{eq}{:}j, \sigma) = \mathsf{T} \quad \text{if } \theta(\sigma) = j \,, \\
 & & \mathit{yld}(\mathsf{eq}{:}j, \sigma) = \mathsf{F} \quad \text{if } \theta(\sigma) \neq j \,, \\
\mathit{eff}(m, \sigma) = \uparrow \quad \text{if } m \notin \mathcal{M}_{\mathsf{irf}} \,, & & \mathit{yld}(m, \sigma) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{irf}} \,, \\
\mathit{eff}(m, \uparrow) = \uparrow \,, & & \mathit{yld}(m, \uparrow) = \mathsf{B} \,.
\end{array}
$$

## 10 Program Synthesis

In this section, we show that, for each $\mathrm{PGA_p}$ program $P$ and polyadic instruction sequence vector $\alpha$, a PGA program $P'$ can be synthesized from $P$ and $\alpha$ such that, for all relevant instruction register file states $\sigma$, $|P'|/_{\mathsf{irf}}\mathcal{IRF}_\sigma$ coincides with $|P|_\sigma(\alpha)$ after the presence of the internal actions $\mathsf{tau}$ and $\mathsf{gl}$ has been concealed.

Let $P$ be a $\mathrm{PGA_p}$ program and $\alpha$ be a polyadic instruction sequence vector. The general idea is that:

[7] We use the following notation for functions: $[\,]$ for the empty function; $[d \mapsto r]$ for the function $f$ with $\mathrm{dom}(f) = \{d\}$ such that $f(d) = r$; and $f \oplus g$ for the function $h$ with $\mathrm{dom}(h) = \mathrm{dom}(f) \cup \mathrm{dom}(g)$ such that, for all $d \in \mathrm{dom}(h)$, $h(d) = f(d)$ if $d \notin \mathrm{dom}(g)$ and $h(d) = g(d)$ otherwise.

– each polyadic instruction sequence in $\alpha$ is translated into a $\mathrm{PGA_p}$ program and an appropriate finite collection of instances of this $\mathrm{PGA_p}$ program in which occurrences of get instructions are replaced by core primitive instructions is generated;
– $P$ and all the generated programs are translated into $\mathrm{PGLC_p}$ programs and these $\mathrm{PGLC_p}$ programs are concatenated;
– the resulting $\mathrm{PGLC_p}$ program is translated into a $\mathrm{PGLD_p}$ program and this program is translated into a PGLD program by replacing all occurrences of the supplementary instructions by core primitive instructions as follows:
  - a switch-over instruction $\#\#\#i$ is replaced by an absolute jump instruction whose effect is a jump to the beginning of an appended instruction sequence whose execution leads, after the state of the instruction register file has been found by a linear search, to a jump to the beginning of the right instance of the $\mathrm{PGA_p}$ program that corresponds to the $i$th polyadic instruction sequence in $\alpha$;
  - a put instruction $\mathsf{\$put}{:}i{:}u$ is simply replaced by the plain basic instruction $\mathsf{irf.put}{:}i{:}u$;
  - a get instruction $\mathsf{\$get}{:}i$ is simply replaced by the absolute jump instruction whose effect is a jump to the position of the instruction itself.

A collection of instances of the $\mathrm{PGA_p}$ program corresponding to a polyadic instruction sequence in $\alpha$ is considered appropriate if it includes all instances that may become the one being executed. $P$ and all the generated programs are translated into $\mathrm{PGLC_p}$ programs because $\mathrm{PGLC_p}$ programs are relocatable: they can be concatenated without disturbing the meaning of jump instructions. The $\mathrm{PGLC_p}$ program resulting from the concatenation is translated into a $\mathrm{PGLD_p}$ program before the supplementary instructions are replaced because the replacement of a switch-over instruction by an absolute jump instruction is simpler than its replacement by a relative jump instruction.

Following the general idea outlined above, we define a function $\mathtt{pgap2pgld}$ that yields, for each $\mathrm{PGA_p}$ program $P$, a function that gives, for each polyadic instruction sequence vector $\alpha$, a PGLD program $P'$ such that, for each relevant instruction register file service state $\sigma$, $|\mathtt{pgld2pga}(P')|\,/_{\mathsf{irf}}\,\mathcal{IRF}_\sigma$ coincides with $|P|_\sigma(\alpha)$ after the presence of the internal actions $\mathsf{tau}$ and $\mathsf{gl}$ has been concealed.

The function $\mathtt{pgap2pgld}$ from the set of all $\mathrm{PGA_p}$ programs to the set all functions from the set of all polyadic instruction sequence vectors to the set of all PGLD programs is defined as follows:

$$
\begin{aligned}
&\mathtt{pgap2pgld}(x)(\alpha) = \\
&\quad \mathit{translate}(\mathtt{pglc2pgld}(\mathit{expand}(x)(\alpha)))\,; \\
&\quad +\mathsf{irf.eq}{:}1\,;\#\#l_{1,1}\,;\ldots;+\mathsf{irf.eq}{:}n'\,;\#\#l_{1,n'}\,; \\
&\qquad \vdots \\
&\quad +\mathsf{irf.eq}{:}1\,;\#\#l_{n,1}\,;\ldots;+\mathsf{irf.eq}{:}n'\,;\#\#l_{n,n'}\,,
\end{aligned}
$$

where $n = \mathrm{len}(\alpha)$, $n' = \max(\mathrm{rng}(\theta))$, the function *expand* from the set of all $\mathrm{PGA_p}$ programs to the set all functions from the set of all polyadic instruction sequence vectors to the set of all PGLCp programs is defined as follows:

$$
\begin{aligned}
&\mathit{expand}(x)(\alpha) = \\
&\quad \mathtt{pga2pglc}(x)\,; \\
&\quad \mathtt{pga2pglc}(\mathit{gen}(\alpha,1,\theta^{-1}(1)))\,;\ldots;\mathtt{pga2pglc}(\mathit{gen}(\alpha,1,\theta^{-1}(n')))\,; \\
&\quad\ \vdots \\
&\quad \mathtt{pga2pglc}(\mathit{gen}(\alpha,n,\theta^{-1}(1)))\,;\ldots;\mathtt{pga2pglc}(\mathit{gen}(\alpha,n,\theta^{-1}(n')))\,,
\end{aligned}
$$

where $n = \mathrm{len}(\alpha)$, $n' = \max(\mathrm{rng}(\theta))$, and the function *gen* from the set of all polyadic instruction sequence vectors, the set of all natural numbers and the set of all instruction register file states to the set of all $\mathrm{PGA_p}$ programs is defined as follows:

$$
\begin{aligned}
&\mathit{gen}(\alpha,i,\sigma) = \mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha,i))[\sigma] && \text{if } 1 \leq i \leq \mathrm{len}(\alpha) \wedge \mathit{valid}(\alpha,i,\sigma)\,,\\
&\mathit{gen}(\alpha,i,\sigma) = \#0 && \text{if } 1 \leq i \leq \mathrm{len}(\alpha) \wedge \neg\ \mathit{valid}(\alpha,i,\sigma)\,,\\
&\mathit{gen}(\alpha,i,\sigma) = ! && \text{if } i = 0 \vee i > \mathrm{len}(\alpha)\,,
\end{aligned}
$$

the function *translate* from the set of all $\mathrm{PGLD_p}$ programs to the set of all PGLD programs is defined as follows:

$$\mathit{translate}(u_1\,;\ldots;u_k) = \psi_1(u_1)\,;\ldots;\psi_1(u_k)\,,$$

where the functions $\psi_j$ from the set of all primitive instructions of $\mathrm{PGLD_p}$ to the set of all primitive instructions of PGLD are defined as follows $(1 \leq j \leq k)$:

$$
\begin{aligned}
&\psi_j(\#\#\#i) &&= \#\#l_i && \text{if } 1 \leq i \leq \mathrm{len}(\alpha)\,,\\
&\psi_j(\#\#\#i) &&= ! && \text{if } i = 0 \vee i > \mathrm{len}(\alpha)\,,\\
&\psi_j(\$\mathsf{put}{:}i{:}u) &&= \mathsf{irf.put}{:}i{:}u\,,\\
&\psi_j(\$\mathsf{get}{:}i) &&= \#\#j\,,\\
&\psi_j(u) &&= u && \text{if } u \text{ is not a supplementary basic instruction}\,,
\end{aligned}
$$

where for each $i \in [1, \mathrm{len}(\alpha)]$:

$$
\begin{aligned}
l_i = {} & \mathrm{len}(\mathtt{pga2pglc}(x)) \\
& + \sum_{h \in [1,\mathrm{len}(\alpha)], h' \in \mathrm{rng}(\theta)} \mathrm{len}(\mathtt{pga2pglc}(\mathit{prj}_{\mathit{pgn}(\alpha,h)}(\mathit{pg}(\alpha,h))[\theta^{-1}(h')])) \\
& + 2 \cdot \max(\mathrm{rng}(\theta)) \cdot (i-1)\,,
\end{aligned}
$$

and for each $i \in [1, \mathrm{len}(\alpha)]$ and $j \in \mathrm{rng}(\theta)$:

$$
\begin{aligned}
l_{i,j} = {} & \mathrm{len}(\mathtt{pga2pglc}(x)) \\
& + \sum_{h \in [1,i-1], h' \in \mathrm{rng}(\theta)} \mathrm{len}(\mathtt{pga2pglc}(\mathit{prj}_{\mathit{pgn}(\alpha,h)}(\mathit{pg}(\alpha,h))[\theta^{-1}(h')])) \\
& + \sum_{h' \in [1,j-1]} \mathrm{len}(\mathtt{pga2pglc}(\mathit{prj}_{\mathit{pgn}(\alpha,i)}(\mathit{pg}(\alpha,i))[\theta^{-1}(h')]))\,.
\end{aligned}
$$

The following theorem states rigorously that, for any $\mathrm{PGA_p}$ program $P$ and polyadic instruction sequence vector $\alpha$, for all relevant instruction register file states $\sigma$, $|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))|\,/_{\mathsf{irf}}\,\mathcal{IRF}_\sigma$ coincides with $|P|_\sigma(\alpha)$ after the presence of the internal actions $\mathsf{tau}$ and $\mathsf{gl}$ has been concealed.

**Theorem 1.** *Let $P$ be a $\mathrm{PGA_p}$ program and $\alpha$ be a polyadic instruction sequence vector, and let $n$ be the highest number occurring in instructions of the form $\$\mathsf{put}{:}i{:}u$ or $\$\mathsf{get}{:}i$ in $P$ or $\alpha$. Take the interval $[1,n]$ for $I$ and the set of all core primitive instructions occurring in instructions of the form $\$\mathsf{put}{:}i{:}u$ in $P$ or $\alpha$ for $U$, and let $\sigma \in IRFS$. Then $\tau_{\mathsf{tau}}(|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))|\,/_{\mathsf{irf}}\,\mathcal{IRF}_\sigma) = \tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha)))$.*

*Proof.* The proof of Theorem 1 follows the same line as the proof of Theorem 1 from [6] given in that paper. Here, we give only a brief outline of the proof of the current theorem.

The proof of this theorem proceeds as follows: (i) we give a set $T$ of closed terms of sort $\mathbf{T}$ with $\tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha))) \in T$, a set $T'$ of closed terms of sort $\mathbf{T}$ with $\tau_{\mathsf{tau}}(|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))|\,/_{\mathsf{irf}}\,\mathcal{IRF}_\sigma) \in T'$, and a bijection $\beta : T \to T'$; (ii) we show that there exists a set $E$ consisting of one derivable equation $p = p'$ for each $p \in T$ such that, for all equations $p = p'$ in $E$:

- $\beta(p) = p''$ is a derivable equation if $p''$ is $p'$ with, for all $q \in T$, all occurrences of $q$ in $p'$ replaced by $\beta(q)$;
- $p' \in T$ only if $p'$ can be rewritten to a $q' \notin T$ using the equations in $E$ from left to right.

This means that $\tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha)))$ and $\tau_{\mathsf{tau}}(|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))|\,/_{\mathsf{irf}}\,\mathcal{IRF}_\sigma)$ denote solutions of the same guarded recursive specification. Because guarded recursive specifications have unique solutions, it follows immediately that $\tau_{\mathsf{tau}}(|\mathtt{pgld2pga}(\mathtt{pgap2pgld}(P)(\alpha))|\,/_{\mathsf{irf}}\,\mathcal{IRF}_\sigma) = \tau_{\mathsf{tau}}(\tau_{\mathsf{gl}}(|P|_\sigma(\alpha)))$. □

In the proof outlined above, an apposite indexing of the closed terms in the sets $T$ and $T'$ facilitates the definition of the bijection $\beta$. Yet, this definition is much more complicated than the definition of the bijection needed in the proof from [6] referred to.

## 11 Conclusions

We have given a formalization of instruction sequences that have been split into fragments. Thread extraction provides the possible joint behaviours of a collection of fragments. We have shown that, for each possible joint behaviour of a collection of fragments, a sequential program can be synthesized from the collection of fragments that exhibits on execution essentially the behaviour in question by interaction with a service. This program synthesis is reminiscent of the service-based variant of projection semantics for program notations used in [7]. However, we consider the program synthesis too complicated to serve a semantical purpose.

In this paper, a fragment in a collection of fragments has two attributes: an ordinal index (position) and a program notation index. We have also mentioned that a nominal index (label) could be an optional attribute. Many other attributes that are relevant in practice can be imagined, e.g. modification date, author, tester, owner, user, and security level. In this paper, we have restricted ourselves to attributes that are indispensable for a theoretical understanding of instruction sequence fragmentation.

The question arises whether the aspects of instruction sequence fragmentation covered in this paper that can be dealt with at the level of threads. An option for future work is to investigate this issue.

## References


1. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
2. J. A. Bergstra and I. Bethke. Predictable and reliable program code: Virtual machine based projection semantics. In J. A. Bergstra and M. Burgess, editors, *Handbook of Network and Systems Administration*. Elsevier, Amsterdam, 2007.
3. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
4. J. A. Bergstra and C. A. Middelburg. Splitting bisimulations and retrospective conditions. *Information and Computation*, 204(7):1083–1138, 2006.
5. J. A. Bergstra and C. A. Middelburg. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
6. J. A. Bergstra and C. A. Middelburg. Instruction sequences with dynamically instantiated instructions. Electronic Report PRG0710, Programming Research Group, University of Amsterdam, November 2007.
7. J. A. Bergstra and C. A. Middelburg. Instruction sequences with indirect jumps. Electronic Report PRG0709, Programming Research Group, University of Amsterdam, November 2007.
8. J. A. Bergstra and C. A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.
9. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
10. W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin, 2000.
11. D. Sannella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 13–30. Springer-Verlag, Berlin, 1999.
12. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, Amsterdam, 1990.

# Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0802] A. Barros and T. Hou, *A Constructive Version of AIP Revisited*, Programming Research Group - University of Amsterdam, 2008.

[PRG0801] J.A. Bergstra and C.A. Middelburg, *Programming an Interpreter Using Molecular Dynamics*, Programming Research Group - University of Amsterdam, 2008.

[PRG0713] J.A. Bergstra, A. Ponse, and M.B. van der Zwaag, *Tuplix Calculus*, Programming Research Group - University of Amsterdam, 2007.

[PRG0712] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Towards a Formalization of Budgets*, Programming Research Group - University of Amsterdam, 2007.

[PRG0711] J.A. Bergstra and C.A. Middelburg, *Program Algebra with a Jump-Shift Instruction*, Programming Research Group - University of Amsterdam, 2007.

[PRG0710] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Dynamically Instantiated Instructions*, Programming Research Group - University of Amsterdam, 2007.

[PRG0709] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences with Indirect Jumps*, Programming Research Group - University of Amsterdam, 2007.

[PRG0708] B. Diertens, *Software (Re-)Engineering with PSF III: an IDE for PSF*, Programming Research Group - University of Amsterdam, 2007.

[PRG0707] J.A. Bergstra and C.A. Middelburg, *An Interface Group for Process Components*, Programming Research Group - University of Amsterdam, 2007.

[PRG0706] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Skew Meadows*, Programming Research Group - University of Amsterdam, 2007.

[PRG0705] J.A. Bergstra, Y. Hirschfeld, and J.V. Tucker, *Meadows*, Programming Research Group - University of Amsterdam, 2007.

[PRG0704] J.A. Bergstra and C.A. Middelburg, *Machine Structure Oriented Control Code Logic (Extended Version)*, Programming Research Group - University of Amsterdam, 2007.

[PRG0703] J.A. Bergstra and C.A. Middelburg, *On the Operating Unit Size of Load/Store Architectures*, Programming Research Group - University of Amsterdam, 2007.

[PRG0702] J.A. Bergstra and A. Ponse, *Interface Groups and Financial Transfer Architectures*, Programming Research Group - University of Amsterdam, 2007.

[PRG0701] J.A. Bergstra, I. Bethke, and M. Burgess, *A Process Algebra Based Framework for Promise Theory*, Programming Research Group - University of Amsterdam, 2007.

[PRG0610] J.A. Bergstra and C.A. Middelburg, *Parallel Processes with Implicit Computational Capital*, Programming Research Group - University of Amsterdam, 2006.

[PRG0609] B. Diertens, *Software (Re-)Engineering with PSF II: from architecture to implementation*, Programming Research Group - University of Amsterdam, 2006.

[PRG0608] A. Ponse and M.B. van der Zwaag, *Risk Assessment for One-Counter Threads*, Programming Research Group - University of Amsterdam, 2006.

[PRG0607] J.A. Bergstra and C.A. Middelburg, *Synchronous Cooperation for Explicit Multi-Threading*, Programming Research Group - University of Amsterdam, 2006.

[PRG0606] J.A. Bergstra and M. Burgess, *Local and Global Trust Based on the Concept of Promises*, Programming Research Group - University of Amsterdam, 2006.

[PRG0605] J.A. Bergstra and J.V. Tucker, *Division Safe Calculation in Totalised Fields*, Programming Research Group - University of Amsterdam, 2006.

[PRG0604] J.A. Bergstra and A. Ponse, *Projection Semantics for Rigid Loops*, Programming Research Group - University of Amsterdam, 2006.

[PRG0603] J.A. Bergstra and I. Bethke, *Predictable and Reliable Program Code: Virtual Machine-based Projection Semantics (submitted for inclusion in the Handbook of Network and Systems Administration)*, Programming Research Group - University of Amsterdam, 2006.

[PRG0602] J.A. Bergstra and A. Ponse, *Program Algebra with Repeat Instruction*, Programming Research Group - University of Amsterdam, 2006.

[PRG0601] J.A. Bergstra and A. Ponse, *Interface Groups for Analytic Execution Architectures*, Programming Research Group - University of Amsterdam, 2006.

[PRG0505] B. Diertens, *Software (Re-)Engineering with PSF*, Programming Research Group - University of Amsterdam, 2005.

[PRG0504] P.H. Rodenburg, *Piecewise Initial Algebra Semantics*, Programming Research Group - University of Amsterdam, 2005.

[PRG0503] T.D. Vu, *Metric Denotational Semantics for BPPA*, Programming Research Group - University of Amsterdam, 2005.

[PRG0502] J.A. Bergstra, I. Bethke, and A. Ponse, *Decision Problems for Pushdown Threads*, Programming Research Group - University of Amsterdam, 2005.

[PRG0501] J.A. Bergstra and A. Ponse, *A Bypass of Cohen's Impossibility Result*, Programming Research Group - University of Amsterdam, 2005.

[PRG0405] J.A. Bergstra and I. Bethke, *An Upper Bound for the Equational Specification of Finite State Services*, Programming Research Group - University of Amsterdam, 2004.

[PRG0404] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Strategic Interleaving*, Programming Research Group - University of Amsterdam, 2004.

[PRG0403] B. Diertens, *A Compiler-projection from PGLEc.MSPio to Parrot*, Programming Research Group - University of Amsterdam, 2004.

[PRG0402] J.A. Bergstra and I. Bethke, *Linear Projective Program Syntax*, Programming Research Group - University of Amsterdam, 2004.

[PRG0401] B. Diertens, *Molecular Scripting Primitives*, Programming Research Group - University of Amsterdam, 2004.

[PRG0302] B. Diertens, *A Toolset for PGA*, Programming Research Group - University of Amsterdam, 2003.

[PRG0301] J.A. Bergstra and P. Walters, *Projection Semantics for Multi-File Programs*, Programming Research Group - University of Amsterdam, 2003.

[PRG0201] I. Bethke and P. Walters, *Molecule-oriented Java Programs for Cyclic Sequences*, Programming Research Group - University of Amsterdam, 2002.

The above reports and more are available through the website: www.science.uva.nl/research/prog/

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

www.science.uva.nl/research/prog/