University of Amsterdam
Programming Research Group

A Protocol for Instruction Stream Processing

J.A. Bergstra
C.A. Middelburg

Report PRG0905 May 2009

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

A Protocol for Instruction Stream Processing

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 107, 1098 XG Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. The behaviour produced by an instruction sequence under execution is a behaviour to be controlled by some execution environment: each step performed actuates the processing of an instruction by the execution environment and a reply returned at completion of the processing determines how the behaviour proceeds. In this paper, we are concerned with the case where the processing takes place remotely. We describe a protocol to deal with the case where the behaviour produced by an instruction sequence under execution leads to the generation of a stream of instructions to be processed and a remote execution unit handles the processing of that stream of instructions.

Keywords: instruction stream processing, thread algebra, process algebra.

1998 ACM Computing Classification: D.2.1, D.2.4, F.1.1, F.3.1.

1 Introduction

The behaviour produced by an instruction sequence under execution is a behaviour to be controlled by some execution environment. It proceeds by performing steps in a sequential fashion. Each step performed actuates the processing of an instruction by the execution environment. A reply returned by the execution environment at completion of the processing of the instruction determines how the behaviour proceeds. Often, the processing of instructions takes place remotely. This means that, on execution of an instruction sequence, a stream of instructions to be processed arises at one place and the processing of that stream of instructions is handled at another place. The objective of the current paper is to bring this phenomenon better into the picture. To achieve this objective, we describe a basic protocol for instruction stream processing that deals with this phenomenon.

The phenomenon sketched above is found if it is impracticable to load the instruction sequence to be executed as a whole. For instance, the storage capacity of the execution unit is too small or the execution unit is too far away. The phenomenon requires special attention because the transmission time of the messages involved in remote processing makes it hard to keep the execution unit busy without intermission. The basic protocol for instruction stream processing is directed towards keeping the execution unit busy. There is no reason to use the

word "remote" in a narrow sense. It is convenient to consider processing remote if it involves message passing with transmission times that are not negligible. In that case, the basic protocol provides a starting-point for studies of basic techniques aimed at increasing processor performance, such as pre-fetching and branch-prediction, at a more abstract level than usual. Therefore, we consider the protocol relevant to the area of computer architectures.

The work presented in this paper is a spin-off of the line of research with which a start was made in [5]. The working hypothesis of that line of research is that instruction sequence is a central notion of computer science. In that line of research, issues concerning the following subjects from the theory of computation have been investigated from the viewpoint that a program is an instruction sequence: semantics of programming languages, expressiveness of programming languages, computability, computational complexity, and performance related matters of instruction sequences (see e.g. [8, 10, 11, 9, 7]). The description of the basic protocol for instruction stream processing starts from the behaviours produced by instruction sequences under execution. By that we abstract from the instruction sequences which produce those behaviours. At the level of abstraction concerned, it is easy to describe how the instruction streams are generated. How instruction streams can be generated efficiently from instruction sequences is another matter.

Threads as considered in BTA (Basic Thread Algebra) are used in this paper to model the behaviours produced by instruction sequences under execution. BTA, introduced under the name BPPA (Basic Polarized Process Algebra) in [5], is a form of process algebra tailored to the description and analysis of the behaviours produced by instruction sequences under execution. General process algebras, such as ACP [4, 2], CCS [15, 17] and CSP [12, 16], are too general for the description and analysis of the behaviours produced by instruction sequences under execution. That is, it is quite awkward to describe and analyse behaviours of this kind using such a general process algebra. However, the behaviours considered in BTA can be viewed as processes that are definable over ACP, see e.g. [6]. This allows for the basic protocol for instruction stream processing to be described using ACP or rather ACP^τ, an extension of ACP which supports abstraction from internal actions.

This paper is organized as follows. First, we give brief summaries of BTA (Section 2) and ACP^τ (Section 3). Next, we show how the behaviours considered in BTA can be viewed as processes that are definable over ACP^τ (Section 4). Then, we describe the basic protocol for instruction stream processing (Section 5) and discuss some conceivable adaptations of the protocol (Section 6). Finally, we make some concluding remarks (Section 7).

2 Thread Algebra

In this section, we review BTA (Basic Thread Algebra). BTA is concerned with behaviours as exhibited by instruction sequences under execution. These behaviours are called threads.

Table 1. 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

In BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of *basic actions* has been given. A thread performs basic actions in a sequential fashion. Upon each basic action performed, a reply from the execution environment of the thread determines how it proceeds. The possible replies are the Boolean values T and F.

To build terms, BTA has the following constants and operators:

- the *deadlock* constant D;
- the *termination* constant S;
- for each $a \in \mathcal{A}$, the binary *postconditional composition* operator $\trianglelefteq a \trianglerighteq$.

We assume that there are infinitely many variables, including x, y, z. Terms are built as usual. We use infix notation for the postconditional composition operator. We introduce *basic action prefixing* as an abbreviation: $a \circ p$, where $a \in \mathcal{A}$ and p is a BTA term, abbreviates $p \trianglelefteq a \trianglerighteq p$.

The thread denoted by a closed term of the form $p \trianglelefteq a \trianglerighteq q$ will first perform a, and then proceed as the thread denoted by p if the reply from the execution environment is T and proceed as the thread denoted by q if the reply from the execution environment is F. The threads denoted by D and S will become inactive and terminate, respectively. This implies that each closed BTA term denotes a thread that will become inactive or terminate after it has performed finitely many basic actions. Infinite threads can be described by guarded recursion.

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 and each t_X is a BTA term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' that contain only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur 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 [3].

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ standing for the unique solution of E for X. The axioms for these constants are given in Table 1. In this table, 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 $\langle Y|E\rangle$. RDP and RSP are actually axiom schemas in which X stands for an arbitrary variable, t_X stands for an arbitrary BTA term, and E stands for an arbitrary guarded recursive specification over BTA. Side conditions are added to restrict what X, t_X and E stand for.

Let $\mathfrak{M}$ be a model of BTA extended with guarded recursion. Then we use the term thread for the elements from the domain of $\mathfrak{M}$, and we denote the interpretations of constants and operators in $\mathfrak{M}$ by the constants and operators themselves. Moreoever, let p be a thread. Then the set of *states* or *residual threads* of p, written $Res(p)$, is inductively defined as follows:

– $p \in Res(p)$;
– if $q \trianglelefteq a \trianglerighteq r \in Res(p)$, then $q \in Res(p)$ and $r \in Res(p)$.

We say that p is a *regular* thread if $Res(p)$ is finite. Being a regular thread coincides with being the solution of a finite guarded recursive specification.

In the sequel, we will make use of a version of BTA in which the following additional assumptions relating to $\mathcal{A}$ are made: (i) a fixed but arbitrary set $\mathcal{F}$ of *foci* has been given; (ii) a fixed but arbitrary set $\mathcal{M}$ of *methods* has been given; (iii) $\mathcal{A} = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. These assumptions are based on the view that the execution environment provides a number of services. Performing a basic action $f.m$ is taken as making a request to the service named f to process command m. As usual, we will write $\mathbb{B}$ for the set $\{\mathsf{T}, \mathsf{F}\}$.

3 Process Algebra

In this section, we review ACP$^\tau$ (Algebra of Communicating Processes with abstraction). This is the process algebra that will be used in Section 4 to make precise what processes are produced by the threads denoted by closed terms of BTA with guarded recursion. For a comprehensive overview of ACP$^\tau$, the reader is referred to [2, 13].

In ACP$^\tau$, it is assumed that a fixed but arbitrary set A of *atomic actions*, with $\tau, \delta \notin \mathsf{A}$, and a fixed but arbitrary commutative and associative function $| : \mathsf{A} \times \mathsf{A} \to \mathsf{A} \cup \{\delta\}$ have been given. The function $|$ is regarded to give the result of synchronously performing any two atomic actions for which this is possible, and to give δ otherwise. In ACP$^\tau$, τ is a special atomic action, called the silent step. The act of performing the silent step is considered unobservable. Because it would otherwise be observable, the silent step is considered an atomic action that cannot be performed synchronously with other atomic actions.

ACP$^\tau$ has the following constants and operators:

– for each $e \in \mathsf{A}$, the *atomic action* constant e;
– the *silent step* constant τ;
– the *deadlock* constant δ;
– the binary *alternative composition* operator $+$;
– the binary *sequential composition* operator $\cdot$;
– the binary *parallel composition* operator $\parallel$;
– the binary *left merge* operator $\lfloor\!\lfloor$;
– the binary *communication merge* operator $|$;
– for each $H \subseteq \mathsf{A}$, the unary *encapsulation* operator ∂_H;
– for each $I \subseteq \mathsf{A}$, the unary *abstraction* operator τ_I.

We assume that there are infinitely many variables, including x, y, z. Terms are built as usual. We use infix notation for the binary operators.

Let p and q be closed ACP$^\tau$ terms, $e \in \mathsf{A}$, and $H, I \subseteq \mathsf{A}$. Intuitively, the constants and operators to build ACP$^\tau$ terms can be explained as follows:

– e first performs atomic action e and next terminates successfully;

Table 2. Axioms of ACP^τ

$x + y = y + x$	A1	$x \cdot \tau = x$	B1
$(x + y) + z = x + (y + z)$	A2	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	B2
$x + x = x$	A3		
$(x + y) \cdot z = x \cdot z + y \cdot z$	A4	$\partial_H(a) = a$ if $a \notin H$	D1
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	A5	$\partial_H(a) = \delta$ if $a \in H$	D2
$x + \delta = x$	A6	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\delta \cdot x = \delta$	A7	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	D4
$x \parallel y = x \between y + y \between x + x \mid y$	CM1	$\tau_I(a) = a$ if $a \notin I$	TI1
$a \between x = a \cdot x$	CM2	$\tau_I(a) = \tau$ if $a \in I$	TI2
$a \cdot x \between y = a \cdot (x \parallel y)$	CM3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	TI3
$(x + y) \between z = x \between z + y \between z$	CM4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	TI4
$a \cdot x \mid b = (a \mid b) \cdot x$	CM5		
$a \mid b \cdot x = (a \mid b) \cdot x$	CM6	$a \mid b = b \mid a$	C1
$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	CM7	$(a \mid b) \mid c = a \mid (b \mid c)$	C2
$(x + y) \mid z = x \mid z + y \mid z$	CM8	$\delta \mid a = \delta$	C3
$x \mid (y + z) = x \mid y + x \mid z$	CM9	$\tau \mid a = \delta$	C4

- τ performs an unobservable atomic action and next terminates successfully;
- δ can neither perform an atomic action nor terminate successfully;
- $p + q$ behaves either as p or as q, but not both;
- $p \cdot q$ first behaves as p and on successful termination of p it next behaves as q;
- $p \parallel q$ behaves as the process that proceeds with p and q in parallel;
- $p \between q$ behaves the same as $p \parallel q$, except that it starts with performing an atomic action of p;
- $p \mid q$ behaves the same as $p \parallel q$, except that it starts with performing an atomic action of p and an atomic action of q synchronously;
- $\partial_H(p)$ behaves the same as p, except that atomic actions from H are blocked;
- $\tau_I(p)$ behaves the same as p, except that atomic actions from I are turned into unobservable atomic actions.

The axioms of ACP^τ are given in Table 2. CM2–CM3, CM5–CM7, C1–C4, D1–D4 and TI1–TI4 are actually axiom schemas in which a, b and c stand for arbitrary constants of ACP^τ, and H and I stand for arbitrary subsets of A. ACP^τ is extended with guarded recursion like BTA.

A *recursive specification* over ACP^τ is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is an ACP^τ term containing only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur in E. Let t be an ACP^τ term without occurrences of abstraction operators containing a variable X. Then an occurrence of X in t is *guarded* if t has a

Table 3. 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

subterm of the form $e \cdot t'$ where $e \in \mathsf{A}$ and t' is a term containing this occurrence of X. Let E be a recursive specification over ACP^τ. Then E is a *guarded recursive specification* if, in each equation $X = t_X \in E$: (i) abstraction operators do not occur in t_X and (ii) all occurrences of variables in t_X are guarded or t_X can be rewritten to such a term using the axioms of ACP^τ in either direction and/or the equations in E except the equation $X = t_X$ from left to right. We are only interested models of ACP^τ in which guarded recursive specifications have unique solutions, such as the models of ACP^τ presented in [2].

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ standing for the unique solution of E for X. The axioms for these constants are given in Table 3. In this table, 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 $\langle Y|E\rangle$. RDP and RSP are actually axiom schemas in which X stands for an arbitrary variable, t_X stands for an arbitrary ACP^τ term, and E stands for an arbitrary guarded recursive specification over ACP^τ. Side conditions are added to restrict what X, t_X and E stand for.

We will write $\sum_{i\in S} p_i$, where $S = \{i_1, \ldots, i_n\}$ and $p_{i_1}, \ldots, p_{i_n}$ are ACP^τ terms, for $p_{i_1} + \ldots + p_{i_n}$. The convention is that $\sum_{i\in S} p_i$ stands for δ if $S = \emptyset$. We will often write X for $\langle X|E\rangle$ if E is clear from the context. It should be borne in mind that, in such cases, we use X as a constant.

4 Process Extraction

In this section, we use ACP^τ with guarded recursion to make mathematically precise what processes are produced by the threads denoted by closed terms of BTA with guarded recursion.

For that purpose, A and $|$ are taken such that the following conditions are satisfied:

$$\mathsf{A} \supseteq \{\mathrm{s}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathbb{B}\} \cup \{\mathrm{r}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathbb{B}\} \cup \{\mathrm{stop}, \mathrm{i}\}$$

and for all $f \in \mathcal{F}$, $d \in \mathcal{M} \cup \mathbb{B}$, and $e \in \mathsf{A}$:

$$\begin{aligned}
&\mathrm{s}_f(d) \mid \mathrm{r}_f(d) = \mathrm{i}\,,\\
&\mathrm{s}_f(d) \mid e = \delta \quad \text{if } e \neq \mathrm{r}_f(d)\,, \qquad \mathrm{stop} \mid e = \delta\,,\\
&e \mid \mathrm{r}_f(d) = \delta \quad \text{if } e \neq \mathrm{s}_f(d)\,, \qquad \mathrm{i} \mid e = \delta\,.
\end{aligned}$$

The *process extraction* operation $|\_|$ determines, for each closed term p of BTA with guarded recursion, a closed term of ACP^τ with guarded recursion that denotes the process produced by the thread denoted by p. The process

Table 4. Defining equations for process extraction operation

$\lvert X\rvert^{\mathrm{c}} = X$
$\lvert\mathsf{S}\rvert^{\mathrm{c}} = \mathrm{stop}$
$\lvert\mathsf{D}\rvert^{\mathrm{c}} = \mathrm{i} \cdot \delta$
$\lvert t_1 \trianglelefteq f.m \trianglerighteq t_2\rvert^{\mathrm{c}} = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot \lvert t_1\rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot \lvert t_2\rvert^{\mathrm{c}})$
$\lvert\langle X \mid E\rangle\rvert^{\mathrm{c}} = \langle X \mid \{Y = \lvert t_Y\rvert^{\mathrm{c}} \mid Y = t_Y \in E\}\rangle$

extraction operation $|\_|$ is defined by $|p| = \tau_{\{\mathrm{stop}\}}(|p|^{\mathrm{c}})$, where $|\_|^{\mathrm{c}}$ is defined by the equations given in Table 4 (for $f \in \mathcal{F}$ and $m \in \mathcal{M}$).

Two atomic actions are involved in performing a basic action of the form $f.m$: one for sending a request to process command m to the service named f and another for receiving a reply from that service upon completion of the processing. For each closed term p of BTA with guarded recursion, $|p|^{\mathrm{c}}$ denotes a process that in the event of termination performs a special termination action just before termination. Abstracting from this termination action yields the process denoted by $|p|$. Some atomic actions introduced above are not used in the definition of the process extraction operation for BTA. Those atomic actions are commonly used in the definition of the process extraction operation for extensions of BTA in which operators for thread-service interaction occur, see e.g. [6].

Let p be a closed term of BTA with guarded recursion. Then we say that $|p|$ is the *process produced by* p.

The process extraction operation preserves the axioms of BTA with guarded recursion. Roughly speaking, this means that the translations of these axioms are derivable from the axioms of ACP^{τ} with guarded recursion. Before we make this fully precise, we have a closer look at the axioms of BTA with guarded recursion.

A proper axiom is an equation or a conditional equation. In Table 1, we do not find proper axioms. Instead of proper axioms, we find axiom schemas without side conditions and axiom schemas with side conditions. The axioms of BTA with guarded recursion are obtained by replacing each axiom schema by all its instances.

We define a function $|\_|$ from the set of all equations and conditional equations of BTA with guarded recursion to the set of all equations of ACP^{τ} with guarded recursion as follows:

$$|t_1 = t_2| \;=\; |t_1| = |t_2|\,,$$
$$|E \Rightarrow t_1 = t_2| \;=\; \{|t_1'| = |t_2'| \mid t_1' = t_2' \in E\} \Rightarrow |t_1| = |t_2|\,.$$

Proposition 1. *Let ϕ be an axiom of* BTA *with guarded recursion. Then $|\phi|$ is derivable from the axioms of* ACP^{τ} *with guarded recursion.*

Proof. The proof is trivial. □

Proposition 1 would go through if no abstraction of the above-mentioned special termination action was made. Notice further that ACP^{τ} without the silent step

constant and the abstraction operator, better known as ACP, would suffice if no abstraction of the special termination action was made.

5 A Protocol for Instruction Stream Processing

In this section, we consider a protocol for instruction stream processing. Before the protocol is described, an extension of ACP is introduced to simplify the description of the protocol.

The following extension of ACP from [1] will be used below: the non-branching conditional operator $:\!\rightarrow$ over $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$. The expression $b :\!\rightarrow p$, is to be read as `if` b `then` p `else` δ. The axioms for the non-branching conditional operator are

$$\mathsf{T} :\!\rightarrow x = x \quad \text{and} \quad \mathsf{F} :\!\rightarrow x = \delta \;.$$

The protocol concerns a system whose main components are an *instruction stream generator* and an *instruction stream execution unit*. The instruction stream generator generates different instruction streams for different threads. This is accomplished by starting it in different states. The general idea of the protocol is that:

- the instruction stream generator generating an instruction stream for a thread of the form $t \trianglelefteq a \trianglerighteq t'$ sends a to the instruction stream execution unit;
- on receipt of a, the instruction stream execution unit gets the execution of a done and sends the reply produced to the instruction stream generator;
- on receipt of the reply, the instruction stream generator proceeds with generating an instruction stream for t if the reply is T and for t' otherwise.

In the case where the thread is S or D, the instruction stream generator sends a special instruction (stop or dead) and the instruction stream execution unit does not send back a reply. The specifics of the protocol considered here are that:

- the instruction stream generator may run ahead of the instruction stream execution unit by not waiting for the receipt of the replies resulting from the execution of instructions that it has sent earlier;
- to ensure that the instruction stream execution unit can handle the run-ahead, each instruction sent by the instruction stream generator is accompanied with the sequence of replies after which the instruction must be executed;
- to correct for replies that have not yet reached the instruction stream generator, each instruction sent is also accompanied with the number of replies received since the last sending of an instruction.

We write $\mathcal{A}'$ for the set $\mathcal{A} \cup \{\mathsf{stop}, \mathsf{dead}\}$. Elements from $\mathcal{A}'$ will loosely be called instructions.

Henceforth, it is assumed that a model of BTA extended with guarded recursion has been given. The restriction of the domain of that model to the regular threads will be denoted by $\mathcal{RT}$.

The functions act, $thrt$, and $thrf$ defined below give, for each thread t different from S and D, the action that t will perform first, the thread with which it will proceed if the reply from the execution environment is T, and the thread with which it will proceed if the reply from the execution environment is F, respectively. The functions $act : \mathcal{RT} \to \mathcal{A}'$, $thrt : \mathcal{RT} \to \mathcal{RT}$, and $thrf : \mathcal{RT} \to \mathcal{RT}$ are defined as follows:

$$\begin{aligned}
&act(\mathsf{S}) = \mathsf{stop} \;, && thrt(\mathsf{S}) = \mathsf{D} \;, && thrf(\mathsf{S}) = \mathsf{D} \;,\\
&act(\mathsf{D}) = \mathsf{dead} \;, && thrt(\mathsf{D}) = \mathsf{D} \;, && thrf(\mathsf{D}) = \mathsf{D} \;,\\
&act(t \trianglelefteq a \trianglerighteq t') = a \;, && thrt(t \trianglelefteq a \trianglerighteq t') = t \;, && thrf(t \trianglelefteq a \trianglerighteq t') = t' \;.
\end{aligned}$$

We write $\mathbb{B}^{\leq n}$, where $n \in \mathbb{N}$, for the set $\{u \in \mathbb{B}^* \mid len(u) \leq n\}$.[1]

It is assumed that a natural number ℓ has been given. The number ℓ is taken for the maximal number of steps that the instruction stream generator may run ahead of the instruction stream execution unit.

The set $\mathcal{IM}$ of *instruction messages* is defined as follows:

$$\mathcal{IM} = [0, \ell] \times \mathbb{B}^{\leq \ell} \times \mathcal{A}' \;.$$

In an instruction message $(n, u, a) \in \mathcal{IM}$:

- n is the number of replies that is acknowledged by the message;
- u is the sequence of replies after which the instruction that is part of the message must be executed;
- a is the instruction that is part of the message.

The instruction stream generator sends instruction messages via an instruction message transmission channel to the instruction stream execution unit. We refer to a succession of transmitted instruction messages as an *instruction stream*. An instruction stream is dynamic by nature, in contradistinction with an instruction sequence.

The set $\mathcal{S}_{\text{ISG}}$ of *instruction stream generator states* is defined as follows:

$$\mathcal{S}_{\text{ISG}} = [0, \ell] \times \mathcal{P}(\mathbb{B}^{\leq \ell+1} \times \mathcal{RT}) \;.$$

In an instruction stream generator state $(n, R) \in \mathcal{S}_{\text{ISG}}$:

- n is the number of replies that has been received by the instruction stream generator since the last acknowledgement of received replies;
- in each $(u, p) \in R$, u is the sequence of replies after which the thread p must be performed.

[1] As usual, we write D^* for the set of all finite sequences with elements from set D and $len(\sigma)$ for the length of finite sequence σ. Moreover, we write ϵ for the empty sequence, d for the sequence having d as sole element, $\sigma\sigma'$ for the concatenation of finite sequences σ and σ', and $tl(\sigma)$ for the tail of finite sequence σ.

The functions *updpm* and *updcr* defined below are used to model the updates of the instruction stream generator state on producing a message and consuming a reply, respectively. The function $updpm : (\mathbb{B}^{\leq \ell} \times \mathcal{RT}) \times \mathcal{S}_{\text{ISG}} \to \mathcal{S}_{\text{ISG}}$ is defined as follows:

$$updpm((u,p),(n,R)) = \begin{cases} (0,(R \setminus \{(u,p)\}) \cup \{(u\mathsf{T}, thrt(p)),(u\mathsf{F}, thrf(p))\}) & \text{if } act(p) \notin \{\mathsf{S},\mathsf{D}\} \\ (0,(R \setminus \{(u,p)\})) & \text{if } act(p) \in \{\mathsf{S},\mathsf{D}\} \end{cases} .$$

The function $updcr : \mathbb{B} \times \mathcal{S}_{\text{ISG}} \to \mathcal{S}_{\text{ISG}}$ is defined as follows:

$$updcr(e,(n,R)) = (n+1, \{(u,p) \mid (eu,p) \in R\}) .$$

The function *sel* defined below is used to model the selection of the sequence of replies and instruction that will be part of the next message produced by the instruction stream generator. The function $sel : \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{RT}) \to \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{RT})$ is defined as follows:

$$sel(R) = \{(u,p) \in R \mid \forall (v,q) \in R \bullet len(u) \leq len(v) \wedge len(u) \leq \ell\} .$$

Notice that $(u,p) \in sel(R)$ and $(v,q) \in R$ only if $u \leq v$. By that depth-first run-ahead is excluded. It happens that the performance of the protocol may change considerably if the function *sel* is replaced by another function.

The set $\mathcal{S}_{\text{ISEU}}$ of *instruction stream execution unit states* is defined as follows:

$$\mathcal{S}_{\text{ISEU}} = [0,\ell] \times \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{A}') .$$

In an instruction stream execution unit state $(n,S) \in \mathcal{S}_{\text{ISEU}}$:

- n is the number of replies for which the instruction stream execution unit still has to receive an acknowledgement;
- in each $(u,a) \in S$, u is the sequence of replies after which the action a must be executed.

The functions *updcm* and *updpr* defined below are used to model the updates of the instruction stream execution unit state on producing a reply and consuming a message, respectively. The function $updcm : \mathcal{IM} \times \mathcal{S}_{\text{ISEU}} \to \mathcal{S}_{\text{ISEU}}$ is defined as follows:

$$updcm((k,u,a),(n,S)) = (n-k, S \cup \{(tl^{n-k}(u),a)\}) .^{2}$$

The function $updpr : \mathbb{B} \times \mathcal{S}_{\text{ISEU}} \to \mathcal{S}_{\text{ISEU}}$ is defined as follows:

$$updpr(e,(n,S)) = (n+1, \{(u,a) \mid (eu,a) \in S\}) .$$

The function *nxt* defined below is used to distinguish between the execution of a basic action $a \in \mathcal{A}$, which leads to a reply, and the execution of S or D, which

[2] $tl^n(u)$ is defined by induction on n as usual: $tl^0(u) = u$ and $tl^{n+1}(u) = tl(tl^{n+1}(u))$.

leads to termination or inaction. The function $nxt : \mathcal{A}' \times \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{A}') \to \mathbb{B}$ is defined as follows:

$$nxt(a, S) = \begin{cases} \mathsf{T} & \text{if } (\epsilon, a) \in S \\ \mathsf{F} & \text{if } (\epsilon, a) \notin S \ . \end{cases}$$

Notice that the set $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ is the set of replies. The instruction stream execution unit sends replies via a reply transmission channel to the instruction stream generator. We refer to a succession of replies as a *reply stream*.

For the purpose of describing the transmission protocol in ACP^τ, A and $|$ are taken such that, in addition to the conditions mentioned at the beginning of Section 4, the following conditions are satisfied:

$$\begin{aligned} \mathsf{A} \supseteq\ & \{\mathrm{s}_i(d) \mid i \in \{1,2\}, d \in \mathcal{IM}\} \cup \{\mathrm{r}_i(d) \mid i \in \{1,2\}, d \in \mathcal{IM}\} \\ \cup\ & \{\mathrm{s}_i(e) \mid i \in \{3,4\}, e \in \mathbb{B}\} \cup \{\mathrm{r}_i(e) \mid i \in \{3,4\}, e \in \mathbb{B}\} \cup \{\mathrm{j}\} \end{aligned}$$

and for all $i \in \{1,2\}$, $j \in \{3,4\}$, $d \in \mathcal{IM}$, $e \in \mathbb{B}$, and $e \in \mathsf{A}$:

$$\begin{aligned} &\mathrm{s}_i(d) \mid \mathrm{r}_i(d) = \mathrm{j} \ , && && \mathrm{s}_j(e) \mid \mathrm{r}_j(e) = \mathrm{j} \ , && \\ &\mathrm{s}_i(d) \mid e = \delta && \text{if } e \neq \mathrm{r}_i(d) \ , && \mathrm{s}_j(e) \mid e = \delta && \text{if } e \neq \mathrm{r}_j(e) \ , \\ &e \mid \mathrm{r}_i(d) = \delta && \text{if } e \neq \mathrm{s}_i(d) \ , && e \mid \mathrm{r}_j(e) = \delta && \text{if } e \neq \mathrm{s}_j(e) \ , \end{aligned}$$

$$\mathrm{j} \mid e = \delta \ .$$

Let $p \in \mathcal{RT}$. Then the process representing the basic protocol for instruction stream processing with regard to thread p is described by

$$\partial_H(\mathit{ISG}_p \parallel \mathit{IMTC} \parallel \mathit{RTC} \parallel \mathit{ISEU}) \ ,$$

where the process ISG_p is recursively specified by the following equations:

$$\begin{aligned} \mathit{ISG}_p &= \mathit{ISG}'_{(0,\{(\epsilon,p)\})} \ , \\ \mathit{ISG}'_{(n,R)} &= \sum_{(u,p) \in sel(R)} \mathrm{s}_1((n, u, act(p))) \cdot \mathit{ISG}'_{updpm((u,p),(n,R))} \\ &\quad + \sum_{e \in \mathbb{B}} \mathrm{r}_4(e) \cdot \mathit{ISG}'_{updcr(e,(n,R))} \\ &\quad \text{(for every } (n,R) \in \mathcal{S}_{\mathrm{ISG}} \text{ with } R \neq \emptyset) \ , \\ \mathit{ISG}'_{(n,\emptyset)} &= \mathrm{j} \\ &\quad \text{(for every } (n,\emptyset) \in \mathcal{S}_{\mathrm{ISG}}) \ , \end{aligned}$$

the process IMTC is recursively specified by the following equation:

$$\mathit{IMTC} = \sum_{d \in \mathcal{IM}} \mathrm{r}_1(d) \cdot \mathrm{s}_2(d) \cdot \mathit{IMTC} \ ,$$

the process RTC is recursively specified by the following equation:

$$RTC = \sum_{e \in \mathbb{B}} \mathrm{r}_3(e) \cdot \mathrm{s}_4(e) \cdot RTC \ ,$$

the process $ISEU$ is recursively specified by the following equations:

$$ISEU = ISEU'_{(0,\emptyset)} \ ,$$

$$\begin{aligned} ISEU'_{(n,S)} = & \sum_{d \in \mathcal{IM}} \mathrm{r}_2(d) \cdot ISEU'_{updcm(d,(n,S))} \\ & + \sum_{f.m \in \mathcal{A}} nxt(f.m, S) :\rightarrow \mathrm{s}_f(m) \cdot ISEU''_{(n,S)} \\ & + nxt(\mathsf{stop}, S) :\rightarrow \mathrm{stop} + nxt(\mathsf{dead}, S) :\rightarrow \mathrm{i} \cdot \delta \end{aligned}$$

(for every $(n, S) \in \mathcal{S}_{\mathrm{ISEU}}$) ,

$$\begin{aligned} ISEU''_{(n,S)} = & \sum_{e \in \mathbb{B}} \mathrm{r}_f(e) \cdot \mathrm{s}_3(e) \cdot ISEU'_{updpr(e,(n,S))} \\ & + \sum_{d \in \mathcal{IM}} \mathrm{r}_2(d) \cdot ISEU''_{updcm(d,(n,S))} \end{aligned}$$

(for every $(n, S) \in \mathcal{S}_{\mathrm{ISEU}}$) ,

and

$$\begin{aligned} H \ = \ & \{\mathrm{s}_i(d) \mid i \in \{1,2\}, d \in \mathcal{IM}\} \cup \{\mathrm{r}_i(d) \mid i \in \{1,2\}, d \in \mathcal{IM}\} \\ \cup \ & \{\mathrm{s}_i(e) \mid i \in \{3,4\}, e \in \mathbb{B}\} \cup \{\mathrm{r}_i(e) \mid i \in \{3,4\}, e \in \mathbb{B}\} \ . \end{aligned}$$

ISG_p is the instruction stream generator generating an instruction sequence for thread p, $IMTC$ is the instruction message transmission channel, RTC is the reply transmission channel, and $ISEU$ is the instruction stream execution unit.

The protocol described above has been designed so as to satisfy the following equation:

$$\tau \cdot |p| = \tau \cdot \tau_{\{\mathrm{j}\}}(\partial_H(ISG_p \parallel IMTC \parallel RTC \parallel ISEU)) \ .$$

We refrain from proving that the protocol satisfies this equation since this paper is first and foremost a conceptual paper.

The transmission channels $IMTC$ and RTC can keep one instruction message and one reply, respectively. The protocol has been designed in such a way that the protocol will also work properly if these channels are replaced by channels with larger capacity and even by channels with unbounded capacity.

6 Adaptations of the Protocol

In this section, we discuss some conceivable adaptations of the protocol described in Section 5.

Consider the case where, for each instruction, it is known what the probability is with which its execution leads to the reply T. This might give reason to adapt the protocol described in Section 5. Suppose that the instruction stream generator states do not only keep the sequences of replies after which threads must be performed, but also the sequences of instructions involved in producing those sequences of replies. Then the probability with which the sequences of replies will happen can be calculated and several conceivable adaptations of the protocol to this probabilistic knowledge are possible by mere changes in the selection of the sequence of replies and instruction that will be part of the next instruction message produced by the instruction stream generator. Among those adaptations are:

- restricting the instruction messages that are produced ahead to the ones where the sequence of replies after which the instruction must be executed will happen with a probability ≥ 0.50, but sticking to breadth-first run-ahead;
- restricting the instruction messages that are produced ahead to the ones where the sequence of replies after which the instruction must be executed will happen with a probability ≥ 0.95, but not sticking to breadth-first run-ahead.

Regular threads can be represented in such a way that it is effectively decidable whether the two threads with which a thread may proceed after performing its first action are identical. Consider the case where threads are represented in the instruction stream generator states in such a way. Then the protocol can be adapted such that no duplication of instruction messages takes place in the cases where the two threads with which a thread possibly proceeds after performing its first action are identical. This can be accomplished by using sequences of elements from $\mathbb{B} \cup \{*\}$, instead of sequences of elements from $\mathbb{B}$, in instruction messages, instruction stream generator states, and instruction stream execution unit states. The occurrence of $*$ at position i in a sequence indicates that the ith reply may be either T or F. The impact of this change on the updates of instruction stream generator states and instruction stream execution unit states is minor.

7 Conclusions

We have described a basic protocol to deal with the phenomenon that, on execution of an instruction sequence, a stream of instructions to be processed arises at one place and the processing of that stream of instructions is handled at another place. By that we have brought this phenomenon better into the picture. We have also discussed some conceivable adaptations of the basic protocol.

The description of the protocol starts from the behaviours produced by instruction sequences under execution. By that we abstract from the instruction sequences which produce those behaviours. How instruction streams can be generated efficiently from instruction sequences is a matter that obviously requires

investigations at a less abstract level. The investigations in question are an option for future work.

We believe that the protocol described in this paper provides a setting in which basic techniques aimed at increasing processor performance, such as prefetching and branch-prediction, can be studied at a more abstract level than usual (cf. [14]). In particular, we think that the protocol can serve as a starting-point for the development of a model with which trade-offs encountered in the design of processor architectures can be clarified. We consider investigations into this matter an interesting option for future work.

References

1. Baeten, J.C.M., Bergstra, J.A.: Process algebra with signals and conditions. In: M. Broy (ed.) Programming and Mathematical Methods, *NATO ASI Series*, vol. F88, pp. 273–323. Springer-Verlag (1992)
2. Baeten, J.C.M., Weijland, W.P.: Process Algebra, *Cambridge Tracts in Theoretical Computer Science*, vol. 18. Cambridge University Press, Cambridge (1990)
3. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (eds.) Proceedings 30th ICALP, *Lecture Notes in Computer Science*, vol. 2719, pp. 1–21. Springer-Verlag (2003)
4. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control **60**(1–3), 109–137 (1984)
5. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
6. Bergstra, J.A., Middelburg, C.A.: Thread algebra with multi-level strategies. Fundamenta Informaticae **71**(2–3), 153–182 (2006)
7. Bergstra, J.A., Middelburg, C.A.: Instruction sequences with dynamically instantiated instructions. Electronic Report PRG0710, Programming Research Group, University of Amsterdam (2007). Available from `http://www.science.uva.nl/research/prog/publications.html`. Also available from `http://arxiv.org/`: `arXiv:0711.4217v3 [cs.PL]`
8. Bergstra, J.A., Middelburg, C.A.: Instruction sequences with indirect jumps. Scientific Annals of Computer Science **17**, 19–46 (2007)
9. Bergstra, J.A., Middelburg, C.A.: Instruction sequences and non-uniform complexity theory. Electronic Report PRG0812, Programming Research Group, University of Amsterdam (2008). Available from `http://www.science.uva.nl/research/prog/publications.html`. Also available from `http://arxiv.org/`: `arXiv:0809.0352v1 [cs.CC]`
10. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic **6**(4), 553–563 (2008)
11. Bergstra, J.A., Ponse, A.: Execution architectures for program algebra. Journal of Applied Logic **5**(1), 170–192 (2007)
12. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM **31**(3), 560–599 (1984)
13. Fokkink, W.J.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin (2000)
14. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, third edn. Morgan Kaufmann, San Francisco (2003)

15. Hennessy, M., Milner, R.: Algebraic laws for non-determinism and concurrency. Journal of the ACM **32**(1), 137–161 (1985)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
17. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

[PRG0904] J.A. Bergstra and C.A. Middelburg, *A Process Calculus with Finitary Comprehended Terms*, Programming Research Group - University of Amsterdam, 2009.

[PRG0903] J.A. Bergstra and C.A. Middelburg, *Transmission Protocols for Instruction Streams*, Programming Research Group - University of Amsterdam, 2009.

[PRG0902] J.A. Bergstra and C.A. Middelburg, *Meadow Enriched ACP Process Algebras*, Programming Research Group - University of Amsterdam, 2009.

[PRG0901] J.A. Bergstra and C.A. Middelburg, *Timed Tuplix Calculus and the Wesseling and van den Berg Equation*, Programming Research Group - University of Amsterdam, 2009.

[PRG0814] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences for the Production of Processes*, Programming Research Group - University of Amsterdam, 2008.

[PRG0813] J.A. Bergstra and C.A. Middelburg, *On the Expressiveness of Single-Pass Instruction Sequences*, Programming Research Group - University of Amsterdam, 2008.

[PRG0812] J.A. Bergstra and C.A. Middelburg, *Instruction Sequences and Non-uniform Complexity Theory*, Programming Research Group - University of Amsterdam, 2008.

[PRG0811] D. Staudt, *A Case Study in Software Engineering with PSF: A Domotics Application*, Programming Research Group - University of Amsterdam, 2008.

[PRG0810] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Poly-Threading*, Programming Research Group - University of Amsterdam, 2008.

[PRG0809] J.A. Bergstra and C.A. Middelburg, *Data Linkage Dynamics with Shedding*, Programming Research Group - University of Amsterdam, 2008.

[PRG0808] B. Diertens, *A Process Algebra Software Engineering Environment*, Programming Research Group - University of Amsterdam, 2008.

[PRG0807] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *Tuplix Calculus Specifications of Financial Transfer Networks*, Programming Research Group - University of Amsterdam, 2008.

[PRG0806] J.A. Bergstra and C.A. Middelburg, *Data Linkage Algebra, Data Linkage Dynamics, and Priority Rewriting*, Programming Research Group - University of Amsterdam, 2008.

[PRG0805] J.A. Bergstra, S. Nolst Trenite, and M.B. van der Zwaag, *UvA Budget Allocatie Model*, Programming Research Group - University of Amsterdam, 2008.

[PRG0804] J.A. Bergstra and C.A. Middelburg, *Thread Algebra for Sequential Poly-Threading*, Programming Research Group - University of Amsterdam, 2008.

[PRG0803] J.A. Bergstra and C.A. Middelburg, *Thread Extraction for Polyadic Instruction Sequences*, Programming Research Group - University of Amsterdam, 2008.

[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/