Instruction Sequences for the Production of Processes

J.A. Bergstra
C.A. Middelburg

Report PRG0814 November 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

Instruction Sequences for the Production of Processes[⋆]

J.A. Bergstra and C.A. Middelburg

Programming Research Group, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, the Netherlands
`J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl`

Abstract. Single-pass instruction sequences under execution are considered to produce behaviours to be controlled by some execution environment. Threads as considered in thread algebra model such behaviours: upon each action performed by a thread, a reply from its execution environment determines how the thread proceeds. Threads in turn can be looked upon as producing processes as considered in process algebra. We show that, by apposite choice of basic instructions, all processes that can only be in a finite number of states can be produced by single-pass instruction sequences.

Keywords: single-pass instruction sequence, process extraction, program algebra, thread algebra, process algebra.

1998 ACM Computing Classification: D.1.4, F.1.1, F.1.2, F.3.2.

1 Introduction

With the current paper, we carry on the line of research with which a start was made in [3]. The working hypothesis of this line of research is that single-pass instruction sequence is a central notion of computer science which merits investigation for its own sake. We take program algebra [3] for the basis of our investigation. Program algebra is a setting suited for investigating single-pass instruction sequences. It does not provide a notation for programs that is intended for actual programming.

The starting-point of program algebra is the perception of a program as a single-pass instruction sequence, i.e. a finite or infinite sequence of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. This perception is simple, appealing, and links up with practice. Single-pass instruction sequences under execution are considered to produce behaviours to be controlled by some execution environment. Threads as considered in basic thread algebra [3] model such behaviours: upon each action performed by a thread, a reply from the execution environment determines how

⋆ This research has been carried out as part of the project *Thread Algebra for Strategic Interleaving*, which is funded by the Netherlands Organisation for Scientific Research (NWO).

the thread proceeds.[1] Threads in turn can be looked upon as producing processes as considered in process algebras such as ACP [1, 7] and CCS [8]. This means that single-pass instruction sequences under execution can be considered to produce such processes.

Process algebra is considered relevant to computer science, as is witnesses by the extent of the work on process algebra in theoretical computer science. This means that there must be programmed systems whose behaviours are taken for processes as considered in process algebra. This has motivated us to investigate the connections between programs and the processes that they produce. In this paper, we investigate those connections starting from the perception of a program as a single-pass instruction sequence.

Regular threads are threads that can only be in a finite number of states. The behaviours of all single-pass instruction sequences considered in program algebra are regular threads and all regular threads can be produced by such single-pass instruction sequences. Regular processes are processes that can only be in a finite number of states. We show in this paper that, by apposite choice of basic instructions, all regular processes can be produced by such single-pass instruction sequences as well.

To obtain this result naturally, we use single-pass instruction sequences with multiple-reply test instructions, which are more general than the test instructions considered in program algebra, and threads with postconditional switching, which is more general than the behavioural counterpart of test instructions considered in basic thread algebra. We show that the result can also be obtained without introducing multiple-reply test instructions and postconditional switching if we assume that the cluster fair abstraction rule (see e.g. [7]) is valid.

Single-pass instruction sequences under execution, and more generally threads, may make use of services such as counters, stacks and Turing tapes. The use operators introduced in [4] are concerned with the effect of services on threads. An interesting aspect of making use of services is that it may turn a regular thread into a non-regular thread. Because non-regular threads produce non-regular processes, this means that single-pass instruction sequences under execution that make use of services may produce non-regular processes. On that account, we add the use operators to basic thread algebra with postconditional switching and make precise what processes are produced by threads that make use of services.

Programs written in an assembly language are finite instruction sequences for which single-pass execution is usually not possible. However, the instruction set of such a program notation may be such that all regular processes can as well be produced by programs written in the program notation. To illustrate this, we show that all regular processes can be produced by programs written in a program notation which is close to existing assembly languages.

This paper is organized as follows. First, we review program algebra and extend it with multiple-reply test instructions (Section 2). Next, we review basic

[1] In [3], basic thread algebra is introduced under the name basic polarized process algebra.

thread algebra, extend it with postconditional switching (Section 3), and use the result to make mathematically precise what threads are produced by the single-pass instruction sequences considered in program algebra with multiple-reply test instructions (Section 4). Then, we review process algebra (Section 5) and use it to make mathematically precise what processes are produced by the threads considered in basic thread algebra with postconditional switching (Section 6). After that, we show that all regular processes can be produced by the single-pass instruction sequences considered in program algebra with multiple-reply test instructions (Section 7). Following this, we extend basic thread algebra with postconditional switching further to threads that make use of services and make precise what processes are produced by such threads (Section 8). After that, we show that all regular processes can also be produced by programs written in a program notation which is close to existing assembly languages (Section 9). Finally, we make some concluding remarks (Section 10).

2 Program Algebra with Multiple-Reply Test Instructions

In this section, we first review PGA (ProGram Algebra) and then extend it with multiple-reply test instructions. All regular processes can be produced by single-pass instruction sequences as considered in PGA extended with multiple-reply test instructions provided use is made of basic instructions of a particular kind. Those basic instructions, which are called process construction instructions, are also introduced.

2.1 Program Algebra

The perception of a program as a single-pass instruction sequence is the starting-point of PGA.

In PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions* has been given. 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 of PGA.

The intuition is that the execution of a basic instruction a produces either T or F at its completion. In the case of a positive test instruction $+a$, a is executed and execution proceeds with the next primitive instruction if T is produced. Otherwise, the next primitive instruction is skipped and execution proceeds with the primitive instruction following the skipped one. If there is no next instruction to be executed, 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, execution always proceeds as if T is produced. The effect of a forward jump

Table 1. 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

instruction $\#l$ is that execution proceeds with the l-th next instruction. If l equals 0 or the l-th next instruction does not exist, deadlock occurs. The effect of the termination instruction ! is that execution terminates.

PGA has the following constants and operators:

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

We assume that there are infinitely many variables, including X, Y, Z. Terms are built as usual. We use infix notation for the concatenation operator and postfix notation for the repetition operator.

A closed PGA term is considered to denote a non-empty, finite or periodic infinite sequence of primitive instructions.[2] Closed PGA terms are considered equal if they denote the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 1. In this table, n stands for an arbitrary natural number greater than 0. For each PGA term P, the term P^n is defined by induction on n as follows: $P^1 = P$ and $P^{n+1} = P\,;P^n$. The *unfolding* equation $X^\omega = X\,;X^\omega$ is derivable. Each closed PGA term is derivably equal to one of the form P or $P\,;Q^\omega$, where P and Q are closed PGA terms in which the repetition operator does not occur.

Notice that PGA2 is actually an axiom schema. Par abus de langage, axiom schemas will be called axioms throughout the paper, with the exception of Section 6.

2.2 Multiple-Reply Test Instructions

We introduce $\mathrm{PGA_{mr}}$, an extension of PGA with multiple-reply test instructions. These additional instructions are like the test instructions of PGA, but cover the case where a natural number greater than zero is produced at the completion of the execution of a basic instruction.

In $\mathrm{PGA_{mr}}$, like in PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of basic instructions has been given. $\mathrm{PGA_{mr}}$ has the primitive instructions of PGA and in addition:

- for each $n \in \mathbb{N}^+$ and $a \in \mathfrak{A}$, a *positive multiple-reply test instruction* $++n{:}a$;[3]

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

[3] We write $\mathbb{N}^+$ for the set $\{n \in \mathbb{N} \mid n > 0\}$.

– for each $n \in \mathbb{N}^+$ and $a \in \mathfrak{A}$, a *negative multiple-reply test instruction* $-\!-n{:}a$.

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

The intuition is that the execution of a basic instruction a produces a natural number greater than zero at its completion. In the case of a positive multiple-reply test instruction $+\!+n{:}a$, a is executed and execution proceeds with the i-th next primitive instruction if a natural number $i \leq n$ is produced. If there is no next instruction to be executed or $i > n$, deadlock occurs. In the case of a negative multiple-reply test instruction $-\!-n{:}a$, execution proceeds with the $n{-}i{+}1$-th next primitive instruction instead of the i-th one if a natural number $i \leq n$ is produced.

For each $a \in \mathfrak{A}$, the instructions $+a$ and $-a$ are considered essentially the same as the instructions $+\!+2{:}a$ and $-\!-2{:}a$, respectively. For that reason, the reply T is identified with the reply 1 and the reply F is identified with the reply 2.

$\mathrm{PGA}_{\mathrm{mr}}$ has a constant u for each $u \in \mathfrak{I}_{\mathrm{mr}}$. The operators of $\mathrm{PGA}_{\mathrm{mr}}$ are the same as the operators as PGA. Likewise, the axioms of $\mathrm{PGA}_{\mathrm{mr}}$ are the same as the axioms as PGA.

The intuition concerning multiple-reply test instructions given above will be made fully precise in Section 4, using an extension of basic thread algebra introduced in Section 3.

2.3 Process Construction and Interaction with Services

Recall that, in $\mathrm{PGA}_{\mathrm{mr}}$, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of basic instructions has been given. In the sequel, we will make use a version of $\mathrm{PGA}_{\mathrm{mr}}$ in which the following additional assumptions relating to $\mathfrak{A}$ are made:

– a fixed but arbitrary set $\mathcal{F}$ of *foci* has been given;
– a fixed but arbitrary set $\mathcal{M}$ of *methods* has been given;
– a fixed but arbitrary set $\mathcal{AA}$ of *atomic actions* has been given;
– $\mathfrak{A}$ consists of:
  - for each $f \in \mathcal{F}$, $m \in \mathcal{M}$, a *program-service interaction instruction* $f.m$;
  - for each $n \in \mathbb{N}^+$, for each $e_1, \ldots, e_n \in \mathcal{AA}$, a *process construction instruction* $\mathsf{ac}(e_1, \ldots, e_n)$.

Each focus plays the role of a name of some service provided by an execution environment that can be requested to process a command. Each method plays the role of a command proper. Executing a basic instruction $f.m$ is taken as making a request to the service named f to process command m.

On execution of a basic instruction $\mathsf{ac}(e_1, \ldots, e_n)$, first a non-deterministic choice between the atomic actions $e_1, \ldots, e_n$ is made and then the chosen atomic action is performed. The reply 1 is produced if e_1 is performed, ..., the reply n is produced if e_n is performed. Basic instructions of this kind are material to produce all regular processes by single-pass instruction sequences.

We will write $\mathrm{PGA}_{\mathrm{mr}}^{\mathrm{pc}}$ for the version of $\mathrm{PGA}_{\mathrm{mr}}$ in which the above-mentioned additional assumptions are made.

The intuition concerning program-service interaction instructions given above will be made fully precise in Section 8, using an extension of basic thread algebra. The intuition concerning process construction instructions given above will be made fully precise in Section 6, using the process algebra introduced in Section 5. It will not be made fully precise using an extension of basic thread algebra because it is considered a basic property of threads that they are deterministic behaviours.

3 Basic Thread Algebra with Postconditional Switching

In this section, we first review BTA (Basic Thread Algebra) and then extend it with postconditional switching. All regular processes can be produced by threads as considered in BTA extended with postconditional switching provided use is made of basic actions of a particular kind. Those basic actions, which are the counterparts of the process construction instructions from $\mathrm{PGA}_{\mathrm{mr}}^{\mathrm{pc}}$, are also introduced.

3.1 Basic Thread Algebra

BTA is concerned with the behaviours that sequential programs exhibit on execution. These behaviours are called threads.

In BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of *basic actions*, with $\mathsf{tau} \notin \mathcal{A}$, has been given. Besides, tau is a special basic action. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. 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 T and F. Performing tau, which is considered performing an internal action, always leads to the reply T.

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 8.

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

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

We assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z. Terms of sort $\mathbf{T}$ are built as usual. We use infix notation for the postconditional composition operator. We introduce *basic action prefixing* as an abbreviation: $a \circ p$ 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.

Table 2. Axiom of BTA

$x \trianglelefteq \mathsf{tau} \trianglerighteq y = x \trianglelefteq \mathsf{tau} \trianglerighteq x$	T1

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

Table 4. Approximation induction principle

$\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	AIP
$\pi_0(x) = \mathsf{D}$	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

BTA has only one axiom. This axiom is given in Table 2. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

Notice that each closed BTA term denotes a thread that will become inactive or terminate after it has performed finitely many 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 of sort $\mathbf{T}$ 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 [2].

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ of sort $\mathbf{T}$ 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 $\langle Y|E\rangle$. 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 what X, t_X and E stand for.

Closed terms that denote the same infinite thread cannot always be proved equal by means of the axioms given in Table 3. We introduce AIP (Approximation Induction Principle) to remedy this. AIP 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 it has performed n actions. In AIP, the approximation up to depth n is phrased in terms of the unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. AIP and the axioms for the projection operators are given in Table 4. In this table, a stands for an arbitrary bascic action from $\mathcal{A}_{\mathsf{tau}}$.

3.2 Postconditional Switching

We introduce $\mathrm{BTA}_{\mathrm{pcs}}$, an extension of BTA with postconditional switching. Postconditional switching is like postconditional composition, but covers the case where the execution environment produces reply values from the set $\mathbb{N}^+$ instead of the set $\{\mathsf{T}, \mathsf{F}\}$. Postconditional switching was first introduced in [6].

In $\mathrm{BTA}_{\mathrm{pcs}}$, like in BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of basic actions, with $\mathsf{tau} \notin \mathcal{A}$, has been given. $\mathrm{BTA}_{\mathrm{pcs}}$ has the constants and operators of BTA and in addition:

– for each $a \in \mathcal{A}_{\mathsf{tau}}$ and $k \in \mathbb{N}^+$, the k-ary *postconditional switch* operator $a \trianglerighteq_k : \underbrace{\mathbf{T} \times \cdots \times \mathbf{T}}_{k \text{ times}} \to \mathbf{T}$.

The thread denoted by a closed terms of the form $a \trianglerighteq_k (p_1, \ldots, p_k)$ will first perform a, and then proceed as the thread denoted by p_1 if the processing of a leads to the reply 1, ..., proceed as the thread denoted by p_k if the processing of a leads to the reply k.

For each $a \in \mathcal{A}_{\mathsf{tau}}$, the operator $\trianglelefteq a \trianglerighteq$ is considered essentially the same as the operator $a \trianglerighteq_2$. For that reason, the reply T is identified with the reply 1 and the reply F is identified with the reply 2.

Without additional assumptions about the set $\mathcal{A}$ of basic actions, axioms S1 and T2 from Table 5 are the only axioms for postconditional switching. Axiom S1 expresses that the operators $\trianglelefteq a \trianglerighteq$ and $a \trianglerighteq_2$ are essentially the same. Like axiom T1, axiom T2 reflects that performing tau always leads to the reply 1.

Guarded recursion can be added to $\mathrm{BTA}_{\mathrm{pcs}}$ as it is added to BTA in Section 3.1.

3.3 Process Construction and Interaction with Services

Recall that, in $\mathrm{BTA}_{\mathrm{pcs}}$, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of basic actions has been given. Like in the case of $\mathrm{PGA}_{\mathrm{mr}}$, we will make use in the sequel of a version of $\mathrm{BTA}_{\mathrm{pcs}}$ in which the following additional assumptions relating to $\mathcal{A}$ are made:

- a fixed but arbitrary set $\mathcal{F}$ of *foci* has been given;
- a fixed but arbitrary set $\mathcal{M}$ of *methods* has been given;
- a fixed but arbitrary set $\mathcal{AA}$ of *atomic actions* has been given;
- $\mathcal{A}$ consists of:
  - for each $f \in \mathcal{F}$ and $m \in \mathcal{M}$, a *thread-service interaction action* $f.m$;
  - for each $n \in \mathbb{N}^+$, for each $e_1, \ldots, e_n \in \mathcal{AA}$, a *process construction action* $\mathsf{ac}(e_1, \ldots, e_n)$.

Like in the case of $\mathrm{PGA}_{\mathrm{mr}}$, performing a basic instruction $f.m$ is taken as making a request to the service named f to process command m.

Like in the case of $\mathrm{PGA}_{\mathrm{mr}}$, on performing a basic action $\mathsf{ac}(e_1, \ldots, e_n)$, first a non-deterministic choice between the atomic actions $e_1, \ldots, e_n$ is made and

Table 5. Axioms for postconditional switching

$x \unlhd a \unrhd y = a \unrhd_2 (x, y)$		S1
$\mathsf{ac}(e_1, \ldots, e_n) \unrhd_k (x_1, \ldots, x_k) = \mathsf{ac}(e_1, \ldots, e_n) \unrhd_n (x_1, \ldots, x_n)$	if $n < k$	S2
$\mathsf{ac}(e_1, \ldots, e_n) \unrhd_k (x_1, \ldots, x_k) = \mathsf{ac}(e_1, \ldots, e_n) \unrhd_n (x_1, \ldots, x_k, \underbrace{\mathsf{D}, \ldots, \mathsf{D}}_{n-k \text{ times}})$	if $n > k$	S3
$\mathsf{tau} \unrhd_k (x_1, \ldots, x_k) = \mathsf{tau} \unrhd_k (\overbrace{x_1, \ldots, x_1}^{k \text{ times}})$		T2

then the chosen atomic action is performed. The reply 1 is produced if e_1 is performed, ..., the reply n is produced if e_n is performed.

In Table 5, axioms are given for the postconditional switching operators which cover the case where the above-mentioned additional assumptions about $\mathcal{A}$ are made. In this table, a stands for an arbitrary basic action from $\mathcal{A}_{\mathsf{tau}}$ and $e_1, \ldots, e_n$ stand for arbitrary atomic actions from $\mathcal{AA}$.

Axioms S2 and S3 stipulate that a thread denoted by a term of the form $\mathsf{ac}(e_1, \ldots, e_n) \unrhd_k (p_1, \ldots, p_k)$ behaves as if it concerns a n-ary postconditional switch if $n \neq k$. The n-ary postconditional switch in question is obtained by removing $p_{n+1}, \ldots, p_k$ if $n < k$, and is obtained by adding D sufficiently many times if $n > k$.

We will write $\mathrm{BTA}_{\mathrm{pcs}}^{\mathrm{pc}}$ for the version of $\mathrm{BTA}_{\mathrm{pcs}}$ in which the above-mentioned additional assumptions are made.

4 Thread Extraction

In this short section, we use $\mathrm{BTA}_{\mathrm{pcs}}$ with guarded recursion to make mathematically precise what threads are produced by the single-pass instruction sequences denoted by closed $\mathrm{PGA}_{\mathrm{mr}}$ terms.

The *thread extraction* operation $|\_|$ determines, for each closed $\mathrm{PGA}_{\mathrm{mr}}$ term P, a closed term of $\mathrm{BTA}_{\mathrm{pcs}}$ with guarded recursion that denotes the thread produced by the single-pass instruction sequence denoted by P. The thread extraction operation is defined by the equations given in Table 6 (for $a \in \mathfrak{A}$, $n \in \mathbb{N}^+$, $l \in \mathbb{N}$, and $u \in \mathfrak{I}_{\mathrm{mr}}$) and the rule that $|\#l \,;\, X| = \mathsf{D}$ if $\#l$ is the beginning of an infinite jump chain. This rule is formalized in e.g. [5].

The equations in Table 6 relating to the primitive instructions of PGA are the equations that have been used to define the thread extraction operation for PGA in most earlier work on PGA (see e.g. [5, 9]). The additional equations relating to multiple-reply test instructions are obvious generalizations of the equations relating to the test instructions of PGA.

Let P be a closed $\mathrm{PGA}_{\mathrm{mr}}$ term. Then we say that $|P|$ is the *thread produced by P*.

Table 6. Defining equations for thread extraction operation

$\lvert a\rvert = a \circ \mathsf{D}$	$\lvert \#l\rvert = \mathsf{D}$
$\lvert a\,;X\rvert = a \circ \lvert X\rvert$	$\lvert \#0\,;X\rvert = \mathsf{D}$
$\lvert +a\rvert = a \circ \mathsf{D}$	$\lvert \#1\,;X\rvert = \lvert X\rvert$
$\lvert +a\,;X\rvert = \lvert X\rvert \trianglelefteq a \trianglerighteq \lvert \#2\,;X\rvert$	$\lvert \#l+2\,;u\rvert = \mathsf{D}$
$\lvert -a\rvert = a \circ \mathsf{D}$	$\lvert \#l+2\,;u\,;X\rvert = \lvert \#l+1\,;X\rvert$
$\lvert -a\,;X\rvert = \lvert \#2\,;X\rvert \trianglelefteq a \trianglerighteq \lvert X\rvert$	
$\lvert {+}{+}n{:}a\rvert = a \circ \mathsf{D}$	$\lvert !\rvert = \mathsf{S}$
$\lvert {+}{+}n{:}a\,;X\rvert = a \trianglerighteq_n (\lvert \#1\,;X\rvert, \ldots, \lvert \#n\,;X\rvert)$	$\lvert !\,;X\rvert = \mathsf{S}$
$\lvert {-}{-}n{:}a\rvert = a \circ \mathsf{D}$	
$\lvert {-}{-}n{:}a\,;X\rvert = a \trianglerighteq_n (\lvert \#n\,;X\rvert, \ldots, \lvert \#1\,;X\rvert)$	

5 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 6 to make precise what processes are produced by the single-pass instruction sequences denoted by closed PGA$^{\mathrm{pc}}_{\mathrm{mr}}$ terms.

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 $\mid : \mathsf{A} \cup \{\tau\} \times \mathsf{A} \cup \{\tau\} \to \mathsf{A} \cup \{\delta\}$, with $\tau \mid e = \delta$ for all $e \in \mathsf{A} \cup \{\tau\}$, have been given. The function $\mid$ 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. We write A_τ for $\mathsf{A} \cup \{\tau\}$.

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 $\mathbin{\|\!\_}$;
- the binary *communication merge* operator $\mid$;
- 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. 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;
- 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;
- $p \mathbin{⌊\!⌊} q$ 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 operators $⌊\!⌊$ and $\mid$ are of an auxiliary nature. They are needed to axiomatize ACP$^\tau$. The axioms of ACP$^\tau$ are given in e.g. [7].

We write $\sum_{i \in S} p_i$, where $S = \{i_1, \ldots, i_n\}$ and $p_{i_1}, \ldots, p_{i_n}$ are ACP$^\tau$ terms, for $p_{i_1} + \ldots + p_{i_n}$. The convention is that $\sum_{i \in S} p_i$ stands for δ if $S = \emptyset$.

A *recursive specification* over ACP$^\tau$ 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$^\tau$ term containing only variables from V. Let t be an ACP$^\tau$ term without occurrences of abstraction operators containing a variable X. Then an occurrence of X in t is *guarded* if t has a 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$^\tau$. 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$^\tau$ in either direction and/or the equations in E except the equation $X = t_X$ from left to right. We only consider models of ACP$^\tau$ in which guarded recursive specifications have unique solutions, such as the models of ACP$^\tau$ presented in [1].

For each guarded recursive specification E and each variable X that occurs in 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 [7].

6 Process Extraction

In this section, we use ACP$^\tau$ with guarded recursion to make mathematically precise what processes are produced by the single-pass instruction sequences denoted by closed PGA$^{\mathrm{pc}}_{\mathrm{mr}}$ terms.

For that purpose, A and $\mid$ are taken such that:

$$\begin{aligned} &\mathcal{AA} \subseteq \mathsf{A}\,, \\ &\mathsf{A} \setminus \mathcal{AA} = \{\mathrm{s}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathbb{N}\} \cup \{\mathrm{r}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathbb{N}\} \\ &\qquad\quad \cup\ \{\mathrm{s}_{\mathrm{serv}}(r) \mid r \in \mathbb{N}\} \cup \{\mathrm{r}_{\mathrm{serv}}(m) \mid m \in \mathcal{M}\} \cup \big\{\mathrm{stop}, \overline{\mathrm{stop}}, \mathrm{stop}^*, \mathrm{i}\big\} \end{aligned}$$

Table 7. Defining equations for process extraction operation

$$
\begin{array}{ll}
|X|' = X & \\
|\mathsf{S}|' = \mathrm{stop} & \\
|\mathsf{D}|' = \mathrm{i} \cdot \delta & \\
|t_1 \trianglelefteq \mathsf{tau} \trianglerighteq t_2|' = \mathrm{i} \cdot \mathrm{i} \cdot |t_1|' & \\
|t_1 \trianglelefteq f.m \trianglerighteq t_2|' = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(1) \cdot |t_1|' + \mathrm{r}_f(2) \cdot |t_2|') & \\
|t_1 \trianglelefteq \mathsf{ac}(e_1, \ldots, e_n) \trianglerighteq t_2|' = e_1 \cdot |t_1|' + e_2 \cdot |t_2|' + \ldots + e_n \cdot |t_2|' & \\
|\,\mathsf{tau} \trianglerighteq_k (t_1, \ldots, t_k)|' = \mathrm{i} \cdot \mathrm{i} \cdot |t_1|' & \\
|\, f.m \trianglerighteq_k (t_1, \ldots, t_k)|' = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(1) \cdot |t_1|' + \ldots + \mathrm{r}_f(k) \cdot |t_k|') & \\
|\,\mathsf{ac}(e_1, \ldots, e_n) \trianglerighteq_k (t_1, \ldots, t_k)|' = e_1 \cdot |t_1|' + \ldots + e_n \cdot |t_n|' & \mathsf{if}\ n \leq k \\
|\,\mathsf{ac}(e_1, \ldots, e_n) \trianglerighteq_k (t_1, \ldots, t_k)|' = & \\
\qquad e_1 \cdot |t_1|' + \ldots + e_k \cdot |t_k|' + e_{k+1} \cdot \mathrm{i} \cdot \delta + \ldots + e_n \cdot \mathrm{i} \cdot \delta & \mathsf{if}\ n > k \\
|\langle X|E\rangle|' = \langle X|\,\{X' = |t_{X'}|' \mid X' = t_{X'} \in E\}\rangle &
\end{array}
$$

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

$$
\begin{array}{llll}
\mathrm{s}_f(d) \mid \mathrm{r}_f(d) = \mathrm{i}\ , & & \mathrm{stop} \mid \overline{\mathrm{stop}} = \mathrm{stop}^*\ , & \\
\mathrm{s}_f(d) \mid e = \delta & \mathsf{if}\ e \neq \mathrm{r}_f(d)\ , & \mathrm{stop} \mid e = \delta & \mathsf{if}\ e \neq \overline{\mathrm{stop}}\ , \\
e \mid \mathrm{r}_f(d) = \delta & \mathsf{if}\ e \neq \mathrm{s}_f(d)\ , & e \mid \overline{\mathrm{stop}} = \delta & \mathsf{if}\ e \neq \mathrm{stop}\ , \\
& & & \\
\mathrm{s}_{\mathrm{serv}}(r) \mid e = \delta\ , & & \mathrm{i} \mid e = \delta\ , & \\
e \mid \mathrm{r}_{\mathrm{serv}}(m) = \delta\ , & & e' \mid e = \delta & \mathsf{if}\ e' \in \mathcal{AA}\ .
\end{array}
$$

The *process extraction* operation $|\_|$ determines, for each closed $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ term p, a closed term of ACP^τ with guarded recursion that denotes the process produced by the thread denoted by p. The process extraction operation $|\_|$ is defined by $|p| = \tau_{\{\mathrm{stop}\}}(|p|')$, where $|\_|'$ is defined by the equations given in Table 7 (for $f \in \mathcal{F}$, $m \in \mathcal{M}$, and $e_1, \ldots, e_n \in \mathcal{AA}$).

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. Performing a basic action of the form $\mathsf{ac}(e_1, \ldots, e_n)$ always gives rise to a non-deterministic choice between n alternatives, where e_i is the first atomic action of the i-th alternatives.

For each closed $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ term p, $|p|'$ denotes a process that will perform a special termination action just before successful termination. Abstracting from this termination action yields the process denoted by $|p|$. In Section 8, $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ is extended with use operators, which are concerned with threads making use of services. The process extraction operation $|\_|$ for $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ is defined here in terms of $|\_|'$ to allow for the process extraction operation for the extension of $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ with use operators to be defined easily.

Some actions introduced above are not used in the definition of the process extraction operation for $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$. Those actions are used in the definition of the process extraction operation for the extension of $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ with use operators.

Let p be a closed $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ term and P be a closed $\mathrm{PGA}^{\mathrm{pc}}_{\mathrm{mr}}$ term. Then we say that $|p|$ is the *process produced by* p and $||P||$ is the *process produced by* P.

The process extraction operation preserves the axioms of $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ 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 $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ with guarded recursion.

A proper axiom is an equation or a conditional equation. In Tables 3 and 5, we do not find proper axioms. Instead of proper axioms, we find axiom schemas without side conditions and axiom schemas with syntactic side conditions. The axioms of $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ 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 $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ with guarded recursion to the set of all equations of ACP^{τ} with guarded recursion as follows:

$$
\begin{aligned}
|t_1 = t_2| &= |t_1| = |t_2| \,, \\
|E \Rightarrow t_1 = t_2| &= \{|t'_1| = |t'_2| \mid t'_1 = t'_2 \in E\} \Rightarrow |t_1| = |t_2| \,.
\end{aligned}
$$

Proposition 1. *Let ϕ be an axiom of $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ 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. However, the expressiveness results for $\mathrm{PGA}^{\mathrm{pc}}_{\mathrm{mr}}$ relating to processes that are presented in Section 7 would not go through. 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.

7 Expressiveness of $\mathbf{PGA}^{\mathbf{pc}}_{\mathbf{mr}}$

In this section, we show that all regular processes can be produced by the single-pass instruction sequences considered in program algebra with multiple-reply test instructions.

We begin by making precise what it means that a thread can only be in a finite number of states. We assume that a fixed but arbitrary model $\mathfrak{M}$ of $\mathrm{BTA}_{\mathrm{pcs}}$ extended with guarded recursion has been given, we use the term thread only 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.

Let p be a thread. Then the set of *states* or *residual threads* of p, written $\mathit{Res}(p)$, is inductively defined as follows:

– $p \in \mathit{Res}(p)$;
– if $q \unlhd a \unrhd r \in \mathit{Res}(p)$, then $p, q \in \mathit{Res}(p)$;
– if $a \unrhd_k (p_1, \ldots, p_k) \in \mathit{Res}(p)$, then $p_1, \ldots, p_k \in \mathit{Res}(p)$.

Let p be a thread and let $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$. Then p is *regular over* $\mathcal{A}'$ if the following conditions are satisfied:

– $\mathit{Res}(p)$ is finite;
– for all $q, r \in \mathit{Res}(p)$ and $a \in \mathcal{A}_{\mathsf{tau}}$, $q \unlhd a \unrhd r \in \mathit{Res}(p)$ implies $a \in \mathcal{A}'$;
– for all $p_1, \ldots, p_k \in \mathit{Res}(p)$ and $a \in \mathcal{A}_{\mathsf{tau}}$, $a \unrhd_k (p_1, \ldots, p_k) \in \mathit{Res}(p)$ implies $a \in \mathcal{A}'$.

We say that p is *regular* if p is regular over $\mathcal{A}_{\mathsf{tau}}$.

We will make use of the fact that being a regular thread coincides with being the solution of a finite guarded recursive specification in which the right-hand sides of the recursion equations are of a restricted form.

A *linear recursive specification* over $\mathrm{BTA}_{\mathrm{pcs}}$ is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$ over $\mathrm{BTA}_{\mathrm{pcs}}$, where each t_X is a term of the form D, S, $Y \unlhd a \unrhd Z$ with $Y, Z \in V$ or $a \unrhd_k (X_1, \ldots, X_k)$ with $X_1, \ldots, X_k \in V$.

Proposition 2. *Let p be a thread and let $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$. Then p is regular over $\mathcal{A}'$ iff there exists a finite linear recursive specification E over* $\mathrm{BTA}_{\mathrm{pcs}}$ *in which only basic actions from $\mathcal{A}'$ occur such that p is the solution of E for some $X \in \mathrm{V}(E)$.*

Proof. This proposition generalizes Theorem 1 from [9] from BTA to $\mathrm{BTA}_{\mathrm{pcs}}$ and from the projective limit model of BTA to an arbitrary model of $\mathrm{BTA}_{\mathrm{pcs}}$. However, the proof of that theorem is applicable to any model of BTA and the adaptations needed to take postconditional switching operators and their interpretations into account are trivial. □

All regular threads over $\mathcal{A}$ can be produced by the single-pass instruction sequences considered in program algebra with multiple-reply test instructions.

Proposition 3. *For each thread p that is regular over $\mathcal{A}$, there exists a closed* $\mathrm{PGA}_{\mathrm{mr}}$ *term P such that p is the thread denoted by $|P|$.*

Proof. This proposition generalizes one direction of Proposition 2 from [9] from PGA to $\mathrm{PGA}_{\mathrm{mr}}$ and from the projective limit model of BTA to an arbitrary model of $\mathrm{BTA}_{\mathrm{pcs}}$. However, the proof of that proposition is applicable to any model of BTA and the adaptations needed to take multiple-reply test instructions and the interpretations of postconditional switching operators into account are trivial. □

We proceed by making precise what it means that a process can only be in a finite number of states. We assume that a fixed but arbitrary model $\mathfrak{M}'$ of ACP^τ with guarded recursion has been given, we use the term process only 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.

Let p be a process. Then the set of *states* or *subprocesses* of p, written $\mathit{Sub}(p)$, is inductively defined as follows:

– $p \in \mathit{Sub}(p)$;
– if $e \cdot q \in \mathit{Sub}(p)$, then $q \in \mathit{Sub}(p)$;
– if $e \cdot q + r \in \mathit{Sub}(p)$, then $q \in \mathit{Sub}(p)$.

Let p be a process and let $\mathsf{A}' \subseteq \mathsf{A}_\tau$. Then p is *regular over* A' if the following conditions are satisfied:

– $\mathit{Sub}(p)$ is finite;
– for all $q \in \mathit{Sub}(p)$ and $e \in \mathsf{A}_\tau$, $e \cdot q \in \mathit{Sub}(p)$ implies $e \in \mathsf{A}'$;
– for all $q, r \in \mathit{Sub}(p)$ and $e \in \mathsf{A}_\tau$, $e \cdot q + r \in \mathit{Sub}(p)$ implies $e \in \mathsf{A}'$.

We say that p is *regular* if p is regular over A_τ.

We will make use of the fact that being a regular process over A coincides with being the solution of a finite guarded recursive specification in which the right-hand sides of the recursion equations are linear terms. *Linearity* of terms is inductively defined as follows:

– δ is linear;
– if $e \in \mathsf{A}_\tau$, then e is linear;
– if $e \in \mathsf{A}_\tau$ and X is a variable, then $e \cdot X$ is linear;
– if t and t' are linear, then $t + t'$ is linear.

A *linear recursive specification* over ACP^τ is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$ over ACP^τ, where each t_X is linear.

Proposition 4. *Let p be a process and let $\mathsf{A}' \subseteq \mathsf{A}$. Then p is regular over A' iff there exists a finite linear recursive specification E over ACP^τ in which only atomic actions from A' occur such that p is the solution of E for some $X \in \mathrm{V}(E)$.*

Proof. The proof follows the same line as the proof of Proposition 2. □

Remark. Proposition 4 is concerned with processes that are regular over A. We can also prove that being a regular process over A_τ coincides with being the solution of a finite linear recursive specification over ACP^τif we assume that the cluster fair abstraction rule [7] holds in the model $\mathfrak{M}'$. However, we do not need this more general result.

All regular processes over $\mathcal{AA}$ can be produced by the single-pass instruction sequences considered in program algebra with multiple-reply test instructions.

Theorem 1. *For each process p that is regular over $\mathcal{AA}$, there exists a closed $\mathrm{PGA}^{\mathrm{pc}}_{\mathrm{mr}}$ term P such that p is the process denoted by $|\lvert P \rvert|$.*

Proof. By Propositions 2, 3 and 4, it is sufficient to show that, for each finite linear recursive specification E over ACP^τ in which only atomic actions from $\mathcal{AA}$ occur, there exists a finite linear recursive specification E' over $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ such that $\langle X|E\rangle = |\langle X|E'\rangle|$ for all $X \in \mathrm{V}(E)$.

Take the finite linear recursive specification E over ACP^τ that consists of the recursion equations

$$X_i = e_{i1} \cdot X_{i1} + \ldots + e_{ik_i} \cdot X_{ik_i} + e'_{i1} + \ldots + e'_{il_i} \ ,$$

where $e_{i1}, \ldots, e_{ik_i}, e'_{i1}, \ldots, e'_{il_i} \in \mathcal{AA}$, for $i \in \{1, \ldots n\}$. Then construct the finite linear recursive specification E' over $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ that consists of the recursion equations

$$X_i = \mathsf{ac}(e_{i1}, \ldots, e_{ik_i}, e'_{i1}, \ldots, e'_{il_i}) \trianglerighteq_{k_i+l_i} (X_{i1}, \ldots, X_{ik_i}, \underbrace{\mathsf{S}, \ldots, \mathsf{S}}_{l_i \text{ times}})$$

for $i \in \{1, \ldots n\}$. It follows immediately from the definition of the process extraction operation that $\langle X|E\rangle = |\langle X|E'\rangle|$ for all $X \in \mathrm{V}(E)$. $\square$

Multiple-reply test instructions and postconditional switching have been introduced because process construction instructions of the form $\mathsf{ac}(e_1, \ldots, e_n)$ with $n > 2$ look to be necessary to obtain this result. However, a similar result can also be obtained for closed $\mathrm{PGA}^{\mathrm{pc}}_{\mathrm{mr}}$ terms in which only basic instructions of the form $\mathsf{ac}(e_1, e_2)$ occur if we assume that the cluster fair abstraction rule [7] holds in the model $\mathfrak{M}'$.

Theorem 2. *Assume that CFAR (Cluster Fair Abstraction Rule) holds in $\mathfrak{M}'$. Let $\mathrm{t} \in \mathcal{AA}$. Then, for each process p that is regular over $\mathcal{AA} \setminus \{\mathrm{t}\}$, there exists a closed $\mathrm{PGA}^{\mathrm{pc}}_{\mathrm{mr}}$ term P in which only basic instructions of the form $\mathsf{ac}(e, \mathrm{t})$ occur such that $\tau \cdot p$ is the process denoted by $\tau \cdot \tau_{\{\mathrm{t}\}}(|\,|P|\,|)$.*

Proof. By Propositions 2, 3 and 4 and the definition of the thread extraction operation, it is sufficient to show that, for each finite linear recursive specification E over ACP^{τ} in which only atomic actions from $\mathcal{AA} \setminus \{\mathrm{t}\}$ occur, there exists a finite linear recursive specification E' over $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ in which only basic actions of the form $\mathsf{ac}(e, \mathrm{t})$ occur such that $\tau \cdot \langle X|E\rangle = \tau \cdot \tau_{\{\mathrm{t}\}}(|\langle X|E'\rangle|)$ for all $X \in \mathrm{V}(E)$.

Take the finite linear recursive specification E over ACP^{τ} that consists of the recursion equations

$$X_i = e_{i1} \cdot X_{i1} + \ldots + e_{ik_i} \cdot X_{ik_i} + e'_{i1} + \ldots + e'_{il_i} \ ,$$

where $e_{i1}, \ldots, e_{ik_i}, e'_{i1}, \ldots, e'_{il_i} \in \mathcal{AA} \setminus \{\mathrm{t}\}$, for $i \in \{1, \ldots n\}$. Then construct the finite linear recursive specification E' over $\mathrm{BTA}^{\mathrm{pc}}_{\mathrm{pcs}}$ that consists of the recursion equations

$$\begin{aligned} X_i = X_{i1} \trianglelefteq \mathsf{ac}(e_{i1}, \mathrm{t}) \trianglerighteq (\ldots (X_{ik_i} \trianglelefteq \mathsf{ac}(e_{ik_i}, \mathrm{t}) \trianglerighteq \\ (\mathsf{S} \trianglelefteq \mathsf{ac}(e'_{i1}, \mathrm{t}) \trianglerighteq (\ldots (\mathsf{S} \trianglelefteq \mathsf{ac}(e'_{il_i}, \mathrm{t}) \trianglerighteq X_i) \ldots))) \ldots) \end{aligned}$$

for $i \in \{1, \ldots n\}$; and the finite linear recursive specification E'' over ACP^{τ} that consists of the recursion equations

$$\begin{array}{llll} X_i & = e_{i1} \cdot X_{i1} + \mathrm{t} \cdot Y_{i2} \ , & Z_{i1} & = e'_{i1} + \mathrm{t} \cdot Z_{i2} \ , \\ Y_{i2} & = e_{i2} \cdot X_{i2} + \mathrm{t} \cdot Y_{i3} \ , & Z_{i2} & = e'_{i2} + \mathrm{t} \cdot Z_{i3} \ , \\ & \vdots & & \vdots \\ Y_{ik_i} & = e_{ik_i} \cdot X_{ik_i} + \mathrm{t} \cdot Z_{i1} \ , & Z_{il_i} & = e'_{il_i} + \mathrm{t} \cdot X_i \ , \end{array}$$

where $Y_{i2}, \ldots, Y_{ik_i}, Z_{i1}, \ldots, Z_{il_i}$ are fresh variables, for $i \in \{1, \ldots n\}$. It follows immediately from the definition of the process extraction operation that

$|\langle X|E'\rangle| = \langle X|E''\rangle$ for all $X \in \mathrm{V}(E)$. Moreover, it follows from CFAR that $\tau\cdot\langle X|E\rangle = \tau\cdot\tau_{\{\mathrm{t}\}}(\langle X|E''\rangle)$ for all $X \in \mathrm{V}(E)$. Hence, $\tau\cdot\langle X|E\rangle = \tau\cdot\tau_{\{\mathrm{t}\}}(|\langle X|E'\rangle|)$ for all $X \in \mathrm{V}(E)$. □

8 Services

An instruction sequence under execution may make use of services. That is, certain instructions may be executed for the purpose of having the behaviour produced by the instruction sequence affected by a service that takes those instructions as commands to be processed. Likewise, a thread may perform certain actions for the purpose of having itself affected by a service that takes those actions as commands to be processed. The processing of an action may involve a change of state of the service and at completion of the processing of the action the service returns a reply value to the thread. The reply value determines how the thread proceeds. In this section, we first review the use operators, which are concerned with threads making such use of services, and then extend the process extraction operation to the use operators. The use operators can be used in combination with the thread extraction operation from Section 4 to describe the behaviour produced by instruction sequences that make use of services.

8.1 Use Operators

A *service* H consists of

- a set S of *states*;
- an *effect* function $\mathit{eff} : \mathcal{M} \times S \to S$;
- a *yield* function $\mathit{yld} : \mathcal{M} \times S \to \mathbb{N}$;
- an *initial state* $s_0 \in S$;

satisfying the following condition:

$$\forall m \in \mathcal{M}, s \in S \bullet (\mathit{yld}(m,s) = 0 \Rightarrow \forall m' \in \mathcal{M} \bullet \mathit{yld}(m', \mathit{eff}(m,s)) = 0)\ .$$

The set S contains the states in which the service 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. By the condition imposed on services, once the service has returned 0 as reply, it keeps returning 0 as reply.

Let $H = (S, \mathit{eff}, \mathit{yld}, s_0)$ be a service and let $m \in \mathcal{M}$. Then the *derived service* of H after processing m, written $\frac{\partial}{\partial m}H$, is the service $(S, \mathit{eff}, \mathit{yld}, \mathit{eff}(m, s_0))$; and the *reply* of H after processing m, written $H(m)$, is $\mathit{yld}(m, s_0)$.

When a thread makes a request to the service to process m:

- if $H(m) \neq 0$, then the request is accepted, the reply is $H(m)$, and the service proceeds as $\frac{\partial}{\partial m}H$;
- if $H(m) = 0$, then the request is rejected and the service proceeds as a service that rejects any request.

Table 8. Axioms for use operators

$\mathsf{S} \mathbin{/_f} H = \mathsf{S}$		U1
$\mathsf{D} \mathbin{/_f} H = \mathsf{D}$		U2
$(x \unlhd \mathsf{tau} \unrhd y) \mathbin{/_f} H = (x \mathbin{/_f} H) \unlhd \mathsf{tau} \unrhd (y \mathbin{/_f} H)$		U3
$(x \unlhd g.m \unrhd y) \mathbin{/_f} H = (x \mathbin{/_f} H) \unlhd g.m \unrhd (y \mathbin{/_f} H)$	$\mathsf{if}\ f \neq g$	U4
$(x \unlhd f.m \unrhd y) \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} \frac{\partial}{\partial m} H)$	$\mathsf{if}\ H(m) = 1$	U5
$(x \unlhd f.m \unrhd y) \mathbin{/_f} H = \mathsf{tau} \circ (y \mathbin{/_f} \frac{\partial}{\partial m} H)$	$\mathsf{if}\ H(m) = 2$	U6
$(x \unlhd f.m \unrhd y) \mathbin{/_f} H = \mathsf{D}$	$\mathsf{if}\ H(m) = 0$	U7
$(x \unlhd \mathsf{ac}(e_1, \ldots, e_n) \unrhd y) \mathbin{/_f} H = (x \mathbin{/_f} H) \unlhd \mathsf{ac}(e_1, \ldots, e_n) \unrhd (y \mathbin{/_f} H)$		U8
$\mathsf{tau} \unrhd_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{tau} \unrhd_k (x_1 \mathbin{/_f} H, \ldots, x_k \mathbin{/_f} H)$		U9
$g.m \unrhd_k (x_1, \ldots, x_k) \mathbin{/_f} H =$ $g.m \unrhd_k (x_1 \mathbin{/_f} H, \ldots, x_k \mathbin{/_f} H)$	$\mathsf{if}\ f \neq g$	U10
$f.m \unrhd_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{tau} \circ (x_i \mathbin{/_f} \frac{\partial}{\partial m} H)$	$\mathsf{if}\ H(\langle m \rangle) = i \land i \in [1, k]$	U11
$f.m \unrhd_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{D}$	$\mathsf{if}\ H(\langle m \rangle) \notin [1, k]$	U12
$\mathsf{ac}(e_1, \ldots, e_n) \unrhd_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{ac}(e_1, \ldots, e_n) \unrhd_k (x_1 \mathbin{/_f} H, \ldots, x_k \mathbin{/_f} H)$		U13
$\pi_n(x \mathbin{/_f} H) = \pi_n(\pi_n(x) \mathbin{/_f} H)$		U14

We introduce the sort $\mathbf{S}$ of *services* and, for each $f \in \mathcal{F}$, the binary *use* operator $\_ \mathbin{/_f} \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. The axioms for these operators are given in Table 8. Intuitively, $p \mathbin{/_f} H$ is the thread that results from processing all actions 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 basic action tau and postconditional composition or postconditional switch is removed in favour of basic action prefixing on the basis of the reply value produced.

We add the use operators to $\mathrm{PGA}^{\mathrm{pc}}_{\mathrm{mr}}$ as well. We will only use the extension in combination with the thread extraction operation $|\_|$ and define $|P \mathbin{/_f} H| = |P| \mathbin{/_f} H$. Hence, $|P \mathbin{/_f} H|$ denotes the thread produced by P if P makes use of H. If H is a service such as an unbounded counter, an unbounded stack or a Turing tape, then a non-regular thread may be produced.

8.2 Extending Process Extraction to the Use Operators

In order to extend the process extraction operation to the use operators, we need an extension of ACP^τ with action renaming operators. The unary action renaming operator ρ_R, for $R : \mathsf{A}_\tau \to \mathsf{A}_\tau$ such that $R(\tau) = \tau$, can be explained as follows: $\rho_R(p)$ behaves as p with each atomic action replaced according to R. The axioms for action renaming are given in [7]. We write $\rho_{e' \mapsto e''}$ for the renaming operator ρ_R with R defined by $R(e') = e''$ and $R(e) = e$ if $e \neq e'$.

We also need to define a set $A_f \subseteq \mathsf{A}$ and a function $R_f : \mathsf{A}_\tau \to \mathsf{A}_\tau$ for each $f \in \mathcal{F}$:

$$A_f = \{\mathrm{s}_f(d) \mid d \in \mathcal{M} \cup \mathbb{N}\} \cup \{\mathrm{r}_f(d) \mid d \in \mathcal{M} \cup \mathbb{N}\}\ ;$$

Table 9. Additional defining equations for process extraction operation

$\lvert t \mathbin{/_f} H\rvert' = \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop},\overline{\mathrm{stop}}\}}(\partial_{A_f}(\lvert t\rvert' \parallel \rho_{R_f}(\lvert H\rvert'))))$
$\lvert H\rvert' = \langle X_H \vert \{X_{H'} = \sum_{m \in \mathcal{M}} \mathrm{r}_{\mathrm{serv}}(m) \cdot \mathrm{s}_{\mathrm{serv}}(H'(m)) \cdot X_{\frac{\partial}{\partial m}H'} + \overline{\mathrm{stop}} \mid H' \in \Delta(H)\}\rangle$

for all $e \in \mathsf{A}_\tau$, $m \in \mathcal{M}$ and $r \in \mathbb{N}$:

$$\begin{aligned}
&R_f(\mathrm{s}_{\mathrm{serv}}(r)) = \mathrm{s}_f(r)\ , \\
&R_f(\mathrm{r}_{\mathrm{serv}}(m)) = \mathrm{r}_f(m)\ , \\
&R_f(e) = e \qquad\qquad \textsf{if } \textstyle\bigwedge_{r' \in \mathbb{N}} e \neq \mathrm{s}_{\mathrm{serv}}(r') \wedge \bigwedge_{m' \in \mathcal{M}} e \neq \mathrm{r}_{\mathrm{serv}}(m')\ .
\end{aligned}$$

The additional defining equations for the process extraction operation concerning the use operators are given in Table 9, where $\Delta(H)$ is inductively defined as follows:

- $H \in \Delta(H)$;
- if $m \in \mathcal{M}$ and $H' \in \Delta(H)$, then $\frac{\partial}{\partial m}H' \in \Delta(H)$.

The extended process extraction operation preserves the axioms for the use operators. Owing to the presence of axiom schemas with semantic side conditions in Table 8, the axioms for the use operators include proper axioms and axioms that have a semantic side condition of the form $H(m) = n$. By that, the precise formulation of the preservation result is somewhat complicated.

Proposition 5.

1. *Let ϕ be a proper axiom for the use operators. Then $\lvert\phi\rvert$ is derivable from the axioms of* ACP$^\tau$ *with action renaming and guarded recursion.*
2. *Let ϕ* if *ψ be an axiom with semantic side condition for the use operators. Then $\lvert\phi\rvert$ is derivable from the axioms of* ACP$^\tau$ *with action renaming and guarded recursion under the assumption that ψ holds.*

Proof. The proof is straightforward. We sketch the proof for axiom U4, writing E_H for $\{X_{H'} = \sum_{m \in \mathcal{M}} \mathrm{r}_{\mathrm{serv}}(m) \cdot \mathrm{s}_{\mathrm{serv}}(H'(m)) \cdot X_{\frac{\partial}{\partial m}H'} + \overline{\mathrm{stop}} \mid H' \in \Delta(H)\}$. By the definition of the process extraction operation, it is sufficient to show that $\lvert(x \unlhd f.m \unrhd y) \mathbin{/_f} H\rvert' = \lvert \mathsf{tau} \circ (x \mathbin{/_f} \frac{\partial}{\partial m}H)\rvert'$ is derivable under the assumption that $H(m) = 1$ holds. In outline, this goes as follows:

$$\begin{aligned}
&\lvert(x \unlhd f.m \unrhd y) \mathbin{/_f} H\rvert' \\
&\ = \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop},\overline{\mathrm{stop}}\}}(\partial_{A_f}(\mathrm{s}_f(m) \cdot (\mathrm{r}_f(1) \cdot x + \mathrm{r}_f(2) \cdot y) \parallel \rho_{R_f}(\langle X_H \vert E_H\rangle)))) \\
&\ = \mathrm{i} \cdot \mathrm{i} \cdot \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop},\overline{\mathrm{stop}}\}}(\partial_{A_f}(x \parallel \rho_{R_f}(\langle X_{\frac{\partial}{\partial m}H} \vert E_H\rangle)))) \\
&\ = \lvert \mathsf{tau} \circ (x \mathbin{/_f} \tfrac{\partial}{\partial m}H)\rvert'\ .
\end{aligned}$$

In the first and third step, we apply defining equations of $\lvert\_\rvert'$. In the second step, we apply axioms of ACP$^\tau$ with action renaming and guarded recursion, and use the assumption that $H(m) = 1$. □

Let P be a closed $\mathrm{PGA}_{\mathrm{mr}}^{\mathrm{pc}}$ term and H be a service. Then $\|P /_f H\|$ denotes the process produced by P if P makes use of H. Instruction sequences that make use of services such as unbounded counters, unbounded stacks or Turing tapes are interesting because they may produce non-regular processes.

9 $\mathbf{PGLD_{mr}}$ Programs and the Use of Boolean Registers

In this section, we show that all regular processes can also be produced by programs written in a program notation which is close to existing assembly languages, and even by programs in which no atomic action occurs more than once. The latter result requires programs that make use of Boolean registers.

9.1 The Program Notation $\mathbf{PGLD_{mr}}$

A hierarchy of program notations rooted in program algebra is introduced in [3]. One program notation that belongs to this hierarchy is PGLD, a very simple program notation which is close to existing assembly languages. It has absolute jump instructions and no explicit termination instruction. Here, we introduce $\mathrm{PGLD_{mr}}$, an extension of PGLD with multiple-reply test instructions.

In $\mathrm{PGLD_{mr}}$, like in $\mathrm{PGA_{mr}}$, it is assumed that there is a fixed but arbitrary finite set of *basic instructions* $\mathfrak{A}$. The primitive instructions of $\mathrm{PGLD_{mr}}$ differ from the primitive instructions of $\mathrm{PGA_{mr}}$ as follows: for each $l \in \mathbb{N}$, there is an *absolute jump instruction* $\#\#l$ instead of a forward jump instruction $\#l$. $\mathrm{PGLD_{mr}}$ programs have the form $u_1; \ldots ; u_k$, where $u_1, \ldots , u_k$ are primitive instructions of $\mathrm{PGLD_{mr}}$.

The effects of all instructions in common with $\mathrm{PGA_{mr}}$ are as in $\mathrm{PGA_{mr}}$ with one difference: if there is no next instruction to be executed, termination occurs. 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.

We define the meaning of $\mathrm{PGLD_{mr}}$ programs by means of a function `pgldmr2pga` from the set of all $\mathrm{PGLD_{mr}}$ programs to the set of all closed $\mathrm{PGA_{mr}}$ terms. This function is defined by

$$\mathtt{pgldmr2pga}(u_1 ; \ldots ; u_k) = (\phi_1(u_1) ; \ldots ; \phi_k(u_k) ; ! ; !)^\omega ,$$

where the auxiliary functions ϕ_j from the set of all primitive instructions of $\mathrm{PGLD_{mr}}$ to the set of all primitive instructions of $\mathrm{PGA_{mr}}$ are defined as follows ($1 \leq j \leq k$):

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

$\mathrm{PGLD_{mr}}$ is as expressive as $\mathrm{PGA_{mr}}$.

Proposition 6. *For each closed* $\mathrm{PGA}_{\mathrm{mr}}$ *term* P*, there exists a* $\mathrm{PGLD}_{\mathrm{mr}}$ *program* P' *such that* $|P| = |\mathtt{pgldmr2pga}(P')|$.

Proof. In [3], a number of functions (called embeddings in that paper) are defined, whose composition gives, for each closed PGA term P, a PGLD program P' such that $|P| = |\mathtt{pgld2pga}(P')|$, where `pgld2pga` is the restriction of `pgldmr2pga` to PGLD programs. The extensions of the above-mentioned embeddings to cover multiple-reply test instructions are trivial because the embeddings change only jump and termination instructions. □

Below, we will write $\mathrm{PGLD}_{\mathrm{mr}}^{\mathrm{pc}}$ for the version of $\mathrm{PGLD}_{\mathrm{mr}}$ in which the additional assumptions relating to $\mathfrak{A}$ mentioned in Section 2.3 are made. As a corollary of Theorem 1 and Proposition 6, we have that all regular processes over $\mathcal{AA}$ can be produced by $\mathrm{PGLD}_{\mathrm{mr}}^{\mathrm{pc}}$ programs.

Corollary 1. *For each process* p *that is regular over* $\mathcal{AA}$*, there exists a* $\mathrm{PGLD}_{\mathrm{mr}}^{\mathrm{pc}}$ *program* P *such that* p *is the process denoted by* $||\mathtt{pgldmr2pga}(P)||$.

9.2 $\mathbf{PGLD_{mr}}$ Programs Acting on Boolean Registers

First, we describe services that make up Boolean registers.

A Boolean register service accepts the following methods:

- a *set to true method* $\mathsf{set{:}T}$;
- a *set to false method* $\mathsf{set{:}F}$;
- a *get method* get.

We write $\mathcal{M}_{\mathsf{BR}}$ for the set $\{\mathsf{set{:}T}, \mathsf{set{:}F}, \mathsf{get}\}$. It is assumed that $\mathcal{M}_{\mathsf{BR}} \subseteq \mathcal{M}$.

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

- $\mathsf{set{:}T}$: the contents of the Boolean register becomes T and the reply is T;
- $\mathsf{set{:}F}$: the contents of the Boolean register becomes F and the reply is F;
- get : nothing changes and the reply is the contents of the Boolean register.

Let $s \in \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$. Then the *Boolean register service* with initial state s, written BR_s, is the service $(\{\mathsf{T}, \mathsf{F}, \mathsf{B}\}, \mathit{eff}, \mathit{eff}, s)$, where the function eff is defined as follows ($b \in \{\mathsf{T}, \mathsf{F}\}$):

$$\begin{aligned}
&\mathit{eff}(\mathsf{set{:}T}, b) = \mathsf{T}\ , && \mathit{eff}(m, b) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{BR}}\ ,\\
&\mathit{eff}(\mathsf{set{:}F}, b) = \mathsf{F}\ , && \mathit{eff}(m, \mathsf{B}) = \mathsf{B}\ .\\
&\mathit{eff}(\mathsf{get}, b) = b\ ,
\end{aligned}$$

Notice that the effect and yield functions of a Boolean register service are the same.

We have that, by making use of Boolean registers, $\mathrm{PGLD}_{\mathrm{mr}}^{\mathrm{pc}}$ programs in which no atomic action from $\mathcal{AA}$ occurs more than once can produce all regular processes over $\mathcal{AA}$.

Theorem 3. *For each process p that is regular over $\mathcal{AA}$, there exists a* $\mathrm{PGLD}^{\mathrm{pc}}_{\mathrm{mr}}$ *program P in which each atomic action from $\mathcal{AA}$ occurs no more than once such that p is the process denoted by $|(\ldots(|\mathtt{pgldmr2pga}(P)|\ /_{\mathsf{br}:1}\ BR_{\mathsf{F}})\ldots /_{\mathsf{br}:k}\ BR_{\mathsf{F}})|$, where k is the length of P.*

Proof. By the proof of Theorem 2 given in Section 7, it is sufficient to show that, for each thread p that is regular over $\mathcal{A}$, there exist a $\mathrm{PGLD}_{\mathrm{mr}}$ program P in which each atomic action from $\mathcal{A}$ occurs no more than once and a $k \in \mathbb{N}^+$ such that p is the thread denoted by $(\ldots(|\mathtt{pgldmr2pga}(P)|\ /_{\mathsf{br}:1}\ BR_{\mathsf{F}})\ldots /_{\mathsf{br}:k}\ BR_{\mathsf{F}})$.

Let p be a thread that is regular over $\mathcal{A}$. We may assume that p is produced by a $\mathrm{PGLD}_{\mathrm{mr}}$ program P' of the following form:

$$
\begin{aligned}
&+a_1 ; \#\#(3 \cdot k_1 + 1) ; \#\#(3 \cdot k'_1 + 1) ;\\
&\quad\vdots\\
&+a_n ; \#\#(3 \cdot k_n + 1) ; \#\#(3 \cdot k'_n + 1) ;\\
&\#\#0 ; \#\#0 ; \#\#0 ; \#\#(3 \cdot n + 4)\ ,
\end{aligned}
$$

where, for each $i \in [1, n]$, $k_i, k'_i \in [0, n-1]$ (cf. the proof of Proposition 2 from [9]). It is easy to see that the $\mathrm{PGLD}_{\mathrm{mr}}$ program P that we are looking for can be obtained by transforming P': by making use of n Boolean registers, P can distinguish between different occurrences of the same basic instruction in P', and in that way simulate P'. □

10 Conclusions

Because process algebra is considered relevant to computer science, there must be programmed systems whose behaviours are taken for processes as considered in process algebra. In that light, we have investigated the connections between programs and the processes that they produce, starting from the perception of a program as a single-pass instruction sequence. We have shown that, by apposite choice of basic instructions, all regular processes can be produced by single-pass instruction sequences as considered in program algebra.

We have also made precise what processes are produced by threads that make use of services. The reason for this is that single-pass instruction sequences under execution are regular threads and regular threads that make use of services such as unbounded counters, unbounded stacks or Turing tapes may produce non-regular processes. An option for future work is to characterize the classes of processes that can be produced by single-pass instruction sequences that make use of such services.

References

1. Baeten, J.C.M., Weijland, W.P.: Process Algebra, *Cambridge Tracts in Theoretical Computer Science*, vol. 18. Cambridge University Press, Cambridge (1990)

2. 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)
3. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51**(2), 125–156 (2002)
4. Bergstra, J.A., Middelburg, C.A.: Thread algebra for strategic interleaving. Formal Aspects of Computing **19**(4), 445–474 (2007)
5. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic **6**(4), 553–563 (2008)
6. Bergstra, J.A., Middelburg, C.A.: Thread algebra for poly-threading. Electronic Report PRG0810, 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:0803.0378v2 [cs.LO]`
7. Fokkink, W.J.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin (2000)
8. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
9. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: A. Beckmann, et al. (eds.) CiE 2006, *Lecture Notes in Computer Science*, vol. 3988, pp. 445–458. Springer-Verlag (2006)

Electronic Reports Series of the Programming Research Group

[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.