University of Amsterdam
Programming Research Group

Program Algebra with a Jump-Shift Instruction

J.A. Bergstra
C.A. Middelburg

Report PRG0711 December 2007

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

Program Algebra with a Jump-Shift Instruction*

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

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

Abstract. We study sequential programs that are instruction sequences with jump-shift instructions in the setting of PGA (ProGram Algebra). Jump-shift instructions preceding a jump instruction increase the position to jump to. The jump-shift instruction is not found in programming practice. Its merit is that the expressive power of PGA extended with the jump-shift instruction, is not reduced if the reach of jump instructions is bounded. This is used to show that there exists a finite-state execution mechanism that by making use of a counter can produce each finite-state thread from some program that is a finite or periodic infinite sequence of instructions from a finite set.

Keywords: program algebra, jump-shift instruction, thread algebra, thread extraction, execution mechanism.

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

1 Introduction

In this paper, we study sequential programs that are instruction sequences with jump-shift instructions. With that we carry on the line of research with which a start was made in [2]. The object pursued with this line of research is the development of a theoretical understanding of possible forms of sequential programs, starting from the simplest form. The view is taken that sequential programs in the simplest form are sequences of instructions. PGA, an algebra of programs in which programs are looked upon as sequences of instructions, is taken for the basis of the development aimed at. The work presented in this paper is part of an investigation of the consequences of small differences in the choice of primitives in the algebra of programs.

In the line of research carried on in this paper, the view is taken that the behaviours of sequential programs under execution are threads as considered in

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

basic thread algebra [2].[3] The experience gained so far leads us to believe that sequential programs are nothing but linear representations of threads.

If n jump-shift instructions precede a jump instruction, they increase the position to jump to by n. This feature of the jump-shift instruction is called jump shifting. It is a programming feature that is not suggested by existing programming practice. Its merit is that the expressive power of PGA extended with the jump-shift instruction, unlike the expressive power of PGA, is not reduced if the reach of jump instructions is bounded. Therefore, we consider a study of programs that are instruction sequences with jump-shift instructions relevant to programming.

We believe that interaction with services provided by an execution environment is inherent in the behaviour of programs under execution. Intuitively, a counter service provides for jump shifting. In this paper, we define the meaning of programs with jump-shift instructions in two different ways. One way covers all programs with jump-shift instructions. The other way covers all programs with jump-shift instructions that contain no other jump instruction than the one whose effect in the absence of preceding jump-shift instructions is a jump to the position of the instruction itself. The latter way corresponds to a finite-state execution mechanism that by making use of a counter produces the behaviour of a program from that program.

A thread proceeds by doing steps in a sequential fashion. A thread may do certain steps only for the sake of getting reply values returned by some service and that way having itself affected by that service. The interaction between behaviours of programs under execution and a counter service referred to above is an interaction with that purpose. In [6], the use mechanism is introduced to allow for such a kind of interaction between threads and services. In this paper, we will use the use mechanism, which has been renamed to thread-service composition, to have behaviours of programs under execution affected by services.

A hierarchy of program notations rooted in PGA is introduced in [2]. Included in this hierarchy are very simple program notations which are close to existing assembly languages up to and including simple program notations that support structured programming by offering a rendering of conditional and loop constructs. However, although they are found in existing assembly programming practice, indirect jump instructions are not considered. In [5], several kinds of indirect jump instructions are considered, including a kind by which recursive method calls can easily be explained. Moreover, dynamic instruction instantiation, a useful programming feature that is not suggested by existing programming practice, is considered in [4].

This paper is organized as follows. First, we review basic thread algebra (Section 2). Next, we review PGA and extend it with the jump-shift instruction (Section 3). After that, we extend basic thread algebra with thread-service composition, introduce a state-based approach to describe services, and give a

[3] In [2], basic thread algebra is introduced under the name basic polarized process algebra. Prompted by the development of thread algebra [6], which is a design on top of it, basic polarized process algebra has been renamed to basic thread algebra.

state-based description of counter services (Section 4). Following this, we revisit the meaning of programs with jump-shift instructions and show that there exists a finite-state execution mechanism that by making use of a counter can produce each finite-state thread from a program that is a finite or periodic infinite sequence of instructions from a finite set (Section 5). Finally, we make some concluding remarks (Section 6).

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution. The behaviours concerned are called *threads*.

In BTA, it is assumed that there is a fixed but arbitrary finite set $\mathcal{A}$ of *basic actions* with $\mathsf{tau} \notin \mathcal{A}$. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. The members of $\mathcal{A}_{\mathsf{tau}}$ are referred to as *actions*.

The intuition is that each basic action performed by a thread is taken as a command to be processed by a service provided by the execution environment of the thread. The processing of a command may involve a change of state of the service concerned. At completion of the processing of the command, the service produces a reply value. This reply is either T or F and is returned to the thread concerned.

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 4.

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

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

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

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

Let p and q be closed terms of sort $\mathbf{T}$ and $a \in \mathcal{A}_{\mathsf{tau}}$. Then $p \trianglelefteq a \trianglerighteq q$ will perform action a, and after that proceed as p if the processing of a leads to the reply T (called a positive reply), and proceed as q if the processing of a leads to the reply F (called a negative reply). The action tau plays a special role. It is a concrete internal action: performing tau will never lead to a state change and always lead to a positive reply, but notwithstanding all that its presence matters.

BTA has only one axiom. This axiom is given in Table 1. Using the abbreviation introduced above, axiom T1 can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

Table 1. Axiom of BTA

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

Table 2. Axioms for guarded recursion

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

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

A *recursive specification* over BTA is a set of recursion equations $\{X = t_X \mid X \in V\}$ where V is a set of variables of sort **T** and each t_X is a BTA term of sort **T** that contains only variables from V. Let E be a recursive specification over BTA. Then we write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. A *solution* of a recursive specification E is a set of threads (in some model of BTA) $\{T_X \mid X \in \mathrm{V}(E)\}$ such that the equations of E hold if, for all $X \in \mathrm{V}(E)$, X stands for T_X.

Let t be a BTA term of sort **T** containing a variable X of sort **T**. Then an occurrence of X in t is *guarded* if t has a subterm of the form $t' \trianglelefteq a \trianglerighteq t''$ containing this occurrence of X. Let E be a recursive specification over BTA. Then E is a *guarded recursive specification* if, in each equation $X = t_X \in E$, all occurrences of variables in t_X are guarded or t_X can be rewritten to such a term using the equations in E from left to right. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [1]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

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

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

Table 3. Approximation induction principle

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

Table 4. Axioms for projection operators

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

such cases, we use X as a constant. We will also use the following abbreviation: a^ω, where $a \in \mathcal{A}_{\mathsf{tau}}$, abbreviates $\langle X|\{X = a \circ X\}\rangle$.

In [3], we show that the threads considered in BTA+REC can be viewed as processes that are definable over ACP [9].

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

AIP is the infinitary conditional equation given in Table 3. Here, following [2], approximation of depth n is phrased in terms of a unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. The axioms for the projection operators are given in Table 4. In this table, a stands for an arbitrary member of $\mathcal{A}_{\mathsf{tau}}$.

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

A *linear recursive specification* over BTA is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$, where each t_X is a term of the form D, S or $Y \trianglelefteq a \trianglerighteq Z$ with $Y, Z \in V$. For each closed term p of sort $\mathbf{T}$ from the language of BTA+REC, there exist a linear recursive specification E and a variable $X \in \mathrm{V}(E)$ such that $p = \langle X|E\rangle$ is derivable from the axioms of BTA+REC.

Below, the interpretations of the constants and operators of BTA+REC in models of BTA+REC are denoted by the constants and operators themselves. Let $\mathcal{A}$ be some model of BTA+REC, and let p be an element from the domain of $\mathcal{A}$. 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 are only interested in models of BTA+REC in which $\mathrm{card}(Res(\langle X|E\rangle)) \leq \mathrm{card}(E)$ for all finite linear recursive specifications E, such as the projective limit model of BTA presented in [1].

3 Program Algebra and the Jump-Shift Instruction

In this section, we first review PGA (ProGram Algebra) and then extend it with the jump-shift instruction, resulting in $\mathrm{PGA}_{\mathrm{js}}$. PGA is an algebra of sequential programs based on the idea that sequential programs are in essence sequences of instructions. PGA provides a program notation for finite-state threads. The jump-shift instruction is not found in programming practice: if one or more jump-shift instructions precede a jump instruction, then each of those jump-shift instructions increases the position to jump to by one.

3.1 PGA

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

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

We write $\mathfrak{I}_{\mathrm{jmp}}$ for the set of all forward jump instructions and $\mathfrak{I}_{\mathrm{PGA}}$ for the set of all primitive instructions of PGA.

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

PGA has the following constants and operators:

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

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

Table 5. Axioms of PGA

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

Table 6. Defining equations for thread extraction operation

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

Table 7. Rule for cyclic jump chains

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

We use infix notation for concatenation and postfix notation for repetition. We also use the notation P^n. For each PGA term P and $n > 0$, the term P^n is defined by induction on n as follows: $P^1 = P$ and $P^{n+1} = P\,;P^n$.

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

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

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

Table 8. Defining formulas for structural congruence predicate

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

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

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

Let E be a finite guarded recursive specification over BTA, and let P_X be a closed PGA term for each $X \in \mathrm{V}(E)$. Let E' be the set of equations that results from replacing in E all occurrences of X by $|P_X|$ for each $X \in \mathrm{V}(E)$. If E' can be obtained by applications of axioms PGA1–PGA4, the defining equations for the thread extraction operation and the rule for cyclic jump chains, then $|P_X|$ is the solution of E for X. Such a finite guarded recursive specification can always be found. Thus, the behaviour of each closed PGA term, is a thread that is definable by a finite guarded recursive specification over BTA. Moreover, each finite guarded recursive specification over BTA can be translated to a closed PGA term of which the behaviour is the solution of the finite guarded recursive specification concerned (cf. Section 4 of [10]).

Closed PGA terms are loosely called PGA *programs*. PGA programs in which the repetition operator do not occur are called *finite* PGA programs.

3.2 The Jump-Shift Instruction

We extend PGA with the jump-shift instruction, resulting in $\mathrm{PGA}_{\mathrm{js}}$.

In $\mathrm{PGA}_{\mathrm{js}}$, like in PGA, it is assumed that there is a fixed but arbitrary finite set $\mathfrak{A}$ of *basic instructions*. $\mathrm{PGA}_{\mathrm{js}}$ has the *primitive instructions* of PGA and in addition:

- a *jump-shift instruction* $\#'$.

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

Table 9. Additional axioms for the jump-shift instruction

$\#' \,;\, \#l = \#l + 1$	JSI1
$\#' \,;\, u = u$	JSI2
$\#'^{\omega} = \#0^{\omega}$	JSI3

Table 10. Additional defining equation for thread extraction operation

$\lvert x\rvert = \lvert x \,;\, \#0\rvert$

If one or more jump-shift instructions precede a jump instruction, then each of those jump-shift instructions increases the position to jump to by one. If one or more jump-shift instructions precede an instruction different from a jump instruction, then those jump-shift instructions have no effect.

$\mathrm{PGA}_{\mathrm{js}}$ has the following constants and operators:

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

The axioms of $\mathrm{PGA}_{\mathrm{js}}$ are the axioms of PGA (Table 5) and in addition the axioms for the jump-shift instruction given in Table 9. In this table, u stands for an arbitrary primitive instruction from $\mathfrak{I}_{\mathrm{PGA}} \setminus \mathfrak{I}_{\mathrm{jmp}}$.

The thread extraction operation of $\mathrm{PGA}_{\mathrm{js}}$ is defined by the same equations and rule as the thread extraction operation of PGA (Tables 6 and 7), on the understanding that u still stands for an arbitrary primitive instruction from $\mathfrak{I}_{\mathrm{PGA}}$, and in addition the equation given in Table 10. The structural congruence predicate of $\mathrm{PGA}_{\mathrm{js}}$ is defined by the same formulas as the structural congruence predicate of PGA (Table 8), on the understanding that $u_1, \ldots, u_n, v_1, \ldots, v_{m+1}$ still stand for arbitrary primitive instructions from $\mathfrak{I}_{\mathrm{PGA}}$.

The additional defining equation $|x| = |x \,;\, \#0|$ for the thread extraction operation expresses that a missing termination instructions leads to deadlock. For all PGA programs P, the equation $|P| = |P \,;\, \#0|$ is derivable from the axioms of PGA and the defining equations for the thread extraction operation of PGA. For all $\mathrm{PGA}_{\mathrm{js}}$ programs P, the equation $|\#l + 2 \,;\, \#' \,;\, P| = |\#l + 2 \,;\, P|$ is derivable from the axioms of $\mathrm{PGA}_{\mathrm{js}}$ and the defining equations of the thread extraction operation of $\mathrm{PGA}_{\mathrm{js}}$.

Obviously, the set of all PGA programs is a proper subset of the set of all $\mathrm{PGA}_{\mathrm{js}}$ programs. Moreover, the thread extraction operation of PGA is the restriction of the thread extraction operation of $\mathrm{PGA}_{\mathrm{js}}$ to the set of all PGA programs. Therefore, we do not distinguish the two thread extraction operations syntactically.

Below, we consider $\mathrm{PGA}_{\mathrm{js}}$ programs that contain no other jump instruction than #0. We will refer to these programs as $\mathrm{PGA}^0_{\mathrm{js}}$ programs.

An interesting point of $\mathrm{PGA}^0_{\mathrm{js}}$ programs is that they make use of a finite set of primitive instructions. It happens that, although PGA programs make

use of an infinite set of primitive instructions, PGA programs do not offer more expressive power than $\mathrm{PGA}^0_{\mathrm{js}}$ programs.

Theorem 1. *Each* PGA *program* P *can be transformed to a* $\mathrm{PGA}^0_{\mathrm{js}}$ *program* P' *such that* $|P| = |P'|$.

Proof. Let P be a PGA program, and let P' be P with, for all $l > 0$, all occurrences of $\#l$ in P replaced by $\#'^l\,;\#0$. Clearly, P' is a $\mathrm{PGA}^0_{\mathrm{js}}$ program. It is easily proved by induction on l that, for each $l > 0$, the equation $\#l = \#'^l\,;\#0$ is derivable from the axioms of $\mathrm{PGA}_{\mathrm{js}}$. From this it follows immediately that the equation $P = P'$ is derivable from the axioms of $\mathrm{PGA}_{\mathrm{js}}$. Consequently, $|P| = |P'|$. □

As a corollary of Theorem 1 and the expressiveness results for PGA in [10], we have that $|\_|$ can produce each finite-state thread from some $\mathrm{PGA}^0_{\mathrm{js}}$ program.

Corollary 1. *For each finite-state thread* p*, there exists a* $\mathrm{PGA}^0_{\mathrm{js}}$ *program* P *such that* $|P| = p$.

This means that each finite-state thread can be produce from a program that is a finite or periodic infinite sequence of instructions from a finite set.

3.3 On Single Pass Execution of Instruction Sequences

The primitive instructions of PGA have been designed to enable single pass execution of instruction sequences. Thread extraction defined in accordance with the idea of single pass execution of instruction sequences should ideally only involve equations of the form $|u\,;x| = p$ where $|x|$ is the only expression of the form $|P|$ that may occur in p. In this section, thread extraction has not been defined in accordance with the idea of single pass execution of instruction sequences. The equations $|{+}a\,;x| = |x| \trianglelefteq a \trianglerighteq |\#2\,;x|$, $|{-}a\,;x| = |\#2\,;x| \trianglelefteq a \trianglerighteq |x|$, and $|\#l+2\,;u\,;x| = |\#l+1\,;x|$ are not of the right form. In Section 5, we define an alternative thread extraction operation for $\mathrm{PGA}^0_{\mathrm{js}}$ programs, which is better in accordance with the idea of single pass execution of instruction sequences. By that thread extraction operation, each $\mathrm{PGA}^0_{\mathrm{js}}$ program is assigned a thread that becomes the behaviour that it exhibits on execution by interaction with a counter service. In Section 4, we introduce thread-service composition, which allows for the intended interaction.

4 Services and Interaction of Threads with Services

In this section, we first extend BTA with thread-service composition, next introduce a state-based approach to describe services, and then use this approach to give a description of counter services. In the current paper, we will only use thread-service composition to have program behaviours affected by some service.

4.1 Thread-Service Composition

A thread may perform certain actions only for the sake of getting reply values returned by services and that way having itself affected by services. We introduce thread-service composition to allow for threads to be affected in this way. Thread-service composition is introduced under the name *use* in [6].

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

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

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

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

A service H can be understood as follows:

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

For each $f \in \mathcal{F}$, we introduce the binary *thread-service composition* operator $\_ /_f \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. Intuitively, $p /_f H$ is the thread that results from processing all actions performed by thread p that are of the form $f.m$ by service H. When an action of the form $f.m$ performed by thread p is processed by service H, it is turned into tau and postconditional composition is removed in favour of action prefixing on the basis of the reply value produced.

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

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

Table 11. Axioms for thread-service composition

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

Table 12. Axioms for abstraction

$\tau_{\mathsf{tau}}(\mathsf{S}) = \mathsf{S}$	TT1
$\tau_{\mathsf{tau}}(\mathsf{D}) = \mathsf{D}$	TT2
$\tau_{\mathsf{tau}}(\mathsf{tau} \circ x) = \tau_{\mathsf{tau}}(x)$	TT3
$\tau_{\mathsf{tau}}(x \trianglelefteq a \trianglerighteq y) = \tau_{\mathsf{tau}}(x) \trianglelefteq a \trianglerighteq \tau_{\mathsf{tau}}(y)$	TT4

The action tau is an internal action whose presence matters. To conceal its presence in the case where it does not matter after all, we also introduce the unary *abstraction* operator $\tau_{\mathsf{tau}} : \mathbf{T} \to \mathbf{T}$.

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

Abstraction can for instance be appropriate in the case where tau arises from turning actions of an auxiliary nature into tau on thread-service composition. Examples of this case will occur in Section 5. Unlike the use mechanism introduced in [6], the use mechanism introduced in [7] incorporates abstraction.

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

The equation $\tau_{\mathsf{tau}}(\mathsf{tau}^\omega) = \mathsf{D}$ is derivable from the axioms of BTA+REC+AIP+ABSTR.

4.2 State-Based Description of Services

We introduce a state-based approach to describe families of services which will be used in Section 4.3. The approach is similar to the approach to describe state machines introduced in [7].

In this approach, a family of services is described by

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

satisfying the following condition:

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

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

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

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

We define, for each $s \in S$, a service H_s in terms of ceff_s and *yld* as follows:

$$H(\alpha \frown \langle m \rangle) = \mathit{yld}(m, \mathit{ceff}_s(\alpha)) \,.$$

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

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

4.3 Counter Services

We give a state-based description of a very simple family of services that constitute a counter. This counter will be used in Section 5 to describe the behaviour of programs in $\mathrm{PGA}_{\mathrm{js}}$.

The counter services accept the following methods:

- a *counter reset method* reset;
- a *counter increment method* incr;
- a *counter decrement method* decr;
- a *counter is-zero method* iszero.

We write $\mathcal{M}_{\mathsf{cnt}}$ for the set $\{\mathsf{reset}, \mathsf{incr}, \mathsf{decr}, \mathsf{iszero}\}$. It is assumed that $\mathcal{M}_{\mathsf{cnt}} \subseteq \mathcal{M}$.

The methods accepted by counter services can be explained as follows:

- reset : the content of the counter is set to zero and the reply is T;
- incr : the content of the counter is incremented by one and the reply is T;
- decr : if the content of the counter is greater than zero, then the content of the counter is decremented by one and the reply is T; otherwise, nothing changes and the reply is F;
- iszero : if the content of the counter equals zero, then nothing changes and the reply is T; otherwise, nothing changes and the reply is F.

Let $s \in \mathbb{N}$. Then we write Cnt_s for the service with initial state s described by $S = \mathbb{N} \cup \{\uparrow\}$, where $\uparrow \notin \mathbb{N}$, and the functions *eff* and *yld* defined as follows ($k \in \mathbb{N}$):

$$
\begin{array}{lll}
\mathit{eff}(\mathsf{reset}, k) = 0 \,, & & \mathit{yld}(\mathsf{reset}, k) = \mathsf{T} \,, \\
\mathit{eff}(\mathsf{incr}, k) = k + 1 \,, & & \mathit{yld}(\mathsf{incr}, k) = \mathsf{T} \,, \\
\mathit{eff}(\mathsf{decr}, 0) = 0 \,, & & \mathit{yld}(\mathsf{decr}, 0) = \mathsf{F} \,, \\
\mathit{eff}(\mathsf{decr}, k + 1) = k \,, & & \mathit{yld}(\mathsf{decr}, k + 1) = \mathsf{T} \,, \\
\mathit{eff}(\mathsf{iszero}, k) = k \,, & & \mathit{yld}(\mathsf{iszero}, 0) = \mathsf{T} \,, \\
 & & \mathit{yld}(\mathsf{iszero}, k + 1) = \mathsf{F} \,, \\
\mathit{eff}(m, k) = \uparrow & \text{if } m \notin \mathcal{M}_{\mathsf{cnt}} \,, & \mathit{yld}(m, k) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{cnt}} \,, \\
\mathit{eff}(m, \uparrow) = \uparrow \,, & & \mathit{yld}(m, \uparrow) = \mathsf{B} \,.
\end{array}
$$

We write Cnt_{init} for Cnt_0.

5 $\mathrm{PGA}^0_{\mathrm{js}}$ Programs Revisited

In this section, we define an alternative thread extraction operation for $\mathrm{PGA}^0_{\mathrm{js}}$ programs, which is in accordance with the idea of single pass execution of instruction sequences. By that thread extraction operation, each $\mathrm{PGA}^0_{\mathrm{js}}$ program is assigned a thread that becomes the behaviour that it exhibits on execution by interaction with a counter service. We also introduce a notion of an execution mechanism. The alternative thread extraction operation induces a finite-state execution mechanism that by making use of a counter can produce each finite-state thread from some $\mathrm{PGA}^0_{\mathrm{js}}$ program.

5.1 Alternative Semantics for $\mathrm{PGA}^0_{\mathrm{js}}$ Programs

When defining the alternative thread extraction operation for $\mathrm{PGA}^0_{\mathrm{js}}$ programs, it is assumed that there is a fixed but arbitrary finite set $\mathcal{F}$ of foci with $\mathsf{cnt} \in \mathcal{F}$ and a fixed but arbitrary finite set $\mathcal{M}$ of methods. Besides, the set $\{f.m \mid f \in \mathcal{F} \setminus \{\mathsf{cnt}\}, m \in \mathcal{M}\}$ is taken as the set $\mathfrak{A}$ of basic instructions. Thereby no real restriction is imposed on the set $\mathfrak{A}$: in the case where the cardinality of $\mathcal{F}$ equals 2, all basic instructions have the same focus and the set $\mathcal{M}$ of methods can be looked upon as the set $\mathfrak{A}$ of basic instructions.

The alternative thread extraction operation $|\_|'$ for $\mathrm{PGA}^0_{\mathrm{js}}$ programs is defined by the equations given in Table 13 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$, $u \in (\mathfrak{I}_{\mathrm{PGA}_{\mathrm{js}}} \setminus \mathfrak{I}_{\mathrm{jmp}}) \cup \{\#0\}$). The thread assigned to a program by this thread extraction operation is not the behaviour that the program exhibits on execution. That behaviour arises from interaction of this thread with a counter service.

The following theorem states rigorously that, for any $\mathrm{PGA}^0_{\mathrm{js}}$ program, the behaviour under execution coincides with the alternative behaviour under execution on interaction with a counter when abstracted from tau.

Table 13. Defining equations for thread extraction operation

$$
\begin{array}{l}
\hline
|x|' = |x\,;\#0|' \\
|a\,;x|' = \mathsf{cnt.reset} \circ (a \circ |x|') \\
|{+}a\,;x|' = \mathsf{cnt.reset} \circ (|x|' \trianglelefteq a \trianglerighteq (\mathsf{cnt.incr} \circ |x|'_{\mathrm{jmp}})) \\
|{-}a\,;x|' = \mathsf{cnt.reset} \circ ((\mathsf{cnt.incr} \circ |x|'_{\mathrm{jmp}}) \trianglelefteq a \trianglerighteq |x|') \\
|\#'\,;x|' = \mathsf{cnt.incr} \circ |x|' \\
|\#0\,;x|' = \mathsf{D} \trianglelefteq \mathsf{cnt.iszero} \trianglerighteq |x|'_{\mathrm{jmp}} \\
|!\,;x|' = \mathsf{S} \\
\\
|\#'\,;x|'_{\mathrm{jmp}} = |x|'_{\mathrm{jmp}} \\
|u\,;x|'_{\mathrm{jmp}} = \mathsf{cnt.decr} \circ (|u\,;x|' \trianglelefteq \mathsf{cnt.iszero} \trianglerighteq |x|'_{\mathrm{jmp}}) \quad \text{if } u \neq \#' \\
\hline
\end{array}
$$

Theorem 2. *For all* $\mathrm{PGA}^0_{\mathrm{js}}$ *programs* P, $|P| = \tau_{\mathsf{tau}}(|P|' \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{\mathrm{init}})$.

Proof. Strictly speaking, we prove this theorem in the algebraic theory obtained by: (i) combining $\mathrm{PGA}_{\mathrm{js}}$ with BTA+REC+AIP+TSC+ABSTR, resulting in a theory with three sorts: a sort **P** of programs, a sort **T** of threads, and a sort **S** of services; (ii) extending the result by taking $|\_|$ and $|\_|'$ for additional operators from sort **P** to sort **T** and taking the semantic equations and rule defining thread extraction and alternative thread extraction for additional axioms. We write $\mathcal{P}$ for the set of all closed terms of sort **P** from the language of the resulting theory and $\mathcal{T}$ for the set of all closed terms of sort **T** from the language of the resulting theory. Moreover, we write $\mathcal{P}^0$ for the set of all closed terms from $\mathcal{P}$ that contain no other jump instructions than #0.

Let

$$
\begin{aligned}
&T = \{|P|, |{\#'}^{i+1}\,;P|, |{\#'}^{i+1}\,;\#0\,;P| \mid i \in \mathbb{N} \wedge P \in \mathcal{P}^0\}\,, \\
&T' = \{\tau_{\mathsf{tau}}(|P|' \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_i), \tau_{\mathsf{tau}}(|P|'_{\mathrm{jmp}} \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{i+1}) \mid i \in \mathbb{N} \wedge P \in \mathcal{P}^0\}\,,
\end{aligned}
$$

and let $\beta : T \to T'$ be the bijection defined by

$$
\begin{aligned}
&\beta(|P|) = \tau_{\mathsf{tau}}(|P|' \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{\mathrm{init}})\,, \\
&\beta(|{\#'}^{i+1}\,;P|) = \tau_{\mathsf{tau}}(|P|' \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{i+1})\,, \\
&\beta(|{\#'}^{i+1}\,;\#0\,;P|) = \tau_{\mathsf{tau}}(|P|'_{\mathrm{jmp}} \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{i+1})\,.
\end{aligned}
$$

For each $p' \in \mathcal{T}$, write $\beta^*(p')$ for p' with, for all $p \in T$, all occurrences of p in p' replaced by $\beta(p)$. Then, it is straightforward to prove that there exists a set E consisting of one derivable equation $p = p'$ for each $p \in T$ such that, for all equations $p = p'$ in E:

- the equation $\beta(p) = \beta^*(p')$ is also derivable;
- if $p' \in T$, then p' can always be rewritten to a $p'' \notin T$ using the equations in E from left to right.

Because $\beta(|P|) = \tau_{\mathsf{tau}}(|P|' \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{\mathrm{init}})$, this means that, for all $P \in \mathcal{P}^0$, $|P|$ and $\tau_{\mathsf{tau}}(|P|' \mathbin{/_{\mathsf{cnt}}} \mathit{Cnt}_{\mathrm{init}})$ are solutions of the same guarded recursive specification.

Because guarded recursive specifications have unique solutions, it follows immediately that, for all $P \in \mathcal{P}^0$, $|P| = \tau_{\mathsf{tau}}(|P|' \,/_{\mathsf{cnt}}\, Cnt_{\mathrm{init}})$. $\square$

As a corollary of Theorem 2 and Corollary 1, we have that $|\_|'$ by making use of a counter can produce each finite-state thread from some $\mathrm{PGA}^0_{\mathrm{js}}$ program.

Corollary 2. *For each finite-state thread p, there exists a $\mathrm{PGA}^0_{\mathrm{js}}$ program P such that $\tau_{\mathsf{tau}}(|P|' \,/_{\mathsf{cnt}}\, Cnt_{\mathrm{init}}) = p$.*

5.2 On Finite-State Execution Mechanisms

Below, we introduce a notion of an execution mechanism. The intuition is that, for a function that assigns a finite-state behaviour to each member of some set of instruction sequences, an execution mechanism is a deterministic behaviour that can produce the behaviour assigned to each of these instruction sequences from the instruction sequence concerned by going through the instructions in the sequence one by one. We believe that there do not exist execution mechanisms that can deal with sequences of instructions from an infinite set. Therefore, we restrict ourselves to finite instruction sets.

Let $\mathfrak{I}$ be a finite set, let $\mathcal{P}$ be a set of non-empty finite or periodic infinite sequences over $\mathfrak{I}$, and let $|\_|$ be a function that assigns a finite-state thread to each member of $\mathcal{P}$. Assume that $\mathsf{pgs} \in \mathcal{F}$, that $\mathsf{hdeq}{:}u \in \mathcal{M}$ for all $u \in \mathfrak{I}$, that $\mathsf{drop} \in \mathcal{M}$, and that basic actions of the form $\mathsf{pgs}.m$ do not occur in $|P|$ for all $P \in \mathcal{P}$. Moreover, for each $P \in \mathcal{P}$, let PGS_P be the service with initial state P described by $S = \mathcal{P} \cup \{\epsilon\} \cup \{\uparrow\}$, where $\uparrow \notin \mathcal{P} \cup \{\epsilon\}$,[5] and the functions *eff* and *yld* defined as follows ($u, u' \in \mathfrak{I}$, $P \in \mathcal{P}$, $Q \in \mathcal{P} \cup \{\epsilon\}$):

$$
\begin{array}{lll}
\mathit{eff}(\mathsf{hdeq}{:}u, Q) = Q\,, & & \mathit{yld}(\mathsf{hdeq}{:}u, \epsilon) = \mathsf{F}\,, \\
 & & \mathit{yld}(\mathsf{hdeq}{:}u, u) = \mathsf{T}\,, \\
 & & \mathit{yld}(\mathsf{hdeq}{:}u, u\,;P) = \mathsf{T}\,, \\
 & & \mathit{yld}(\mathsf{hdeq}{:}u, u') = \mathsf{F} \quad \text{if } u \neq u'\,, \\
 & & \mathit{yld}(\mathsf{hdeq}{:}u, u'\,;P) = \mathsf{F} \quad \text{if } u \neq u'\,, \\
\mathit{eff}(\mathsf{drop}, \epsilon) = \epsilon\,, & & \mathit{yld}(\mathsf{drop}, \epsilon) = \mathsf{F}\,, \\
\mathit{eff}(\mathsf{drop}, u) = \epsilon\,, & & \mathit{yld}(\mathsf{drop}, u) = \mathsf{T}\,, \\
\mathit{eff}(\mathsf{drop}, u\,;P) = P\,, & & \mathit{yld}(\mathsf{drop}, u\,;P) = \mathsf{T}\,, \\
\mathit{eff}(m, Q) = \uparrow \quad \text{if } m \notin \mathcal{M}_{\mathsf{pgs}}\,, & & \mathit{yld}(m, Q) = \mathsf{B} \quad \text{if } m \notin \mathcal{M}_{\mathsf{pgs}}\,, \\
\mathit{eff}(m, \uparrow) = \uparrow\,, & & \mathit{yld}(m, \uparrow) = \mathsf{B}\,.
\end{array}
$$

Then an *execution mechanism* for $|\_|$ is a thread p such that $\tau_{\mathsf{tau}}(p \,/_{\mathsf{pgs}}\, PGS_P) = |P|$ for all $P \in \mathcal{P}$. An execution mechanism is called a *finite-state* execution mechanism if it is a finite-state thread.

In order to execute an instruction sequence P, an execution mechanism makes use of the service PGS_P to go through that the instructions in that sequence one by one. The methods accepted by this service can be explained as follows:

[5] We write ϵ for the empty sequence.

– $\mathsf{hdeq}{:}u$: if there is an instruction sequence left and its first instruction is u, then nothing changes and the reply is T; otherwise, nothing changes and the reply is F;
– drop : if there is an instruction sequence left, then its first instruction is dropped and the reply is T; otherwise, nothing changes and the reply is F.

Notice that the service does not have to hold an infinite object: there exists an adequate finite representation for each finite or periodic infinite sequence of instructions.

It is easy to see that there exists a finite-state execution mechanism for the thread extraction operation $|\_|'$ for $\mathrm{PGA}^0_{\mathrm{js}}$ programs. From this and Corollary 2, it follows immediately that there exists a finite-state execution mechanism that by making use of a counter can produce each finite-state thread from some program that is a finite or periodic infinite sequence of instructions from a finite set.

We also have that there does not exist a finite-state execution mechanism that by itself can produce each finite-state thread from a program that is a finite or periodic infinite sequence of instructions from a finite set.

Theorem 3. *Let $\mathfrak{I}$ be a finite set, let $\mathcal{P}$ be a set of non-empty finite or periodic infinite sequences over $\mathfrak{I}$, and let $|\_|$ be a function that assigns a finite-state thread to each member of $\mathcal{P}$. Assume that, for each finite-state thread p, there exists a $P \in \mathcal{P}$ such that $|P| = p$. Then there does not exist a finite-state execution mechanism for $|\_|$.*

Proof. Suppose that there exists a finite-state execution mechanism, say p_{exec}. Let n be the number of states of p_{exec}. Consider the thread T_0 defined by the guarded recursive specification consisting of the following equations:

$$\begin{aligned}
T_i &= T_{i+1} \trianglelefteq a \trianglerighteq T'_{i+1,0} \text{ for } i \in [0,n] \;,\\
T_{n+1} &= \mathsf{S} \;,\\
T'_{i+1,i'} &= b \circ T'_{i+1,i'+1} \qquad \text{for } i \in [0,n], i' \in [0,i] \;,\\
T'_{i+1,i+1} &= c \circ T'_{i+1,0} \;.
\end{aligned}$$

Let P be a member of $\mathcal{P}$ from which p_{exec} can produce T_0. Notice that T_0 performs a at least once and at most $n+1$ times after each other. Suppose that T_0 has performed a for the jth time when the reply F is returned, while at that stage p_{exec} has gone through the first k_j instructions of P. Moreover, write P_j for what is left of P after its first k_j instructions have been dropped. Then p_{exec} still has to produce $T'_{j,0}$ from P_j. For each $j \in [1, n+1]$, a k_j as above can be found. Let j_0 be the unique $j \in [1, n+1]$ such that $k_{j'} \leq k_j$ for all $j' \in [1, n+1]$. Regardless the number of times T_0 has performed a when the reply F is returned, p_{exec} must eventually have dropped the first k_{j_0} instructions of P. For each of the $n+1$ possible values of j, p_{exec} must be in a different state when P_{j_0} is left, because the thread that p_{exec} still has to produce is different. However, this is impossible with n states. □

In the light of Theorem 3, Corollary 2 can be considered a positive result: a finite-state execution mechanism that makes use of a counter is sufficient. However, this result is reached at the expense of an extremely inefficient way of representing jumps. We do not see how to improve on the linear representation of jumps. With a logarithmic representation, for instance, we expect that a counter will not do.

Theorem 3 is actually a generalization of Theorem 4 from [8] adapted to the current setting.

The hierarchy of program notations rooted in program algebra introduced in [2] includes a program notation, called PGLS, that supports structured programming by offering a rendering of conditional and loop constructs instead of (unstructured) jump instructions. Like $\mathrm{PGA}^0_{\mathrm{js}}$, PGLS has a finite set of primitive instructions. Like for $\mathrm{PGA}^0_{\mathrm{js}}$ programs, there exists a finite-state execution mechanism that by making use of a counter can produce the behaviour of each PGLS program. However, PGLS programs offer less expressive power than PGA programs (see Section 9 of [2]). Therefore, PGLS is unsuited to show that there exists a finite-state execution mechanism that by making use of a counter can produce each finite-state thread from some program that is a finite or periodic infinite sequence of instructions from a finite set.

6 Conclusions

We have studied sequential programs that are instruction sequences with jump-shift instructions. We have defined the meaning of the programs concerned in two different ways which both involve the extraction of threads. One way covers only programs with jump-shift instructions that contain no other jump instruction than the one whose effect in the absence of preceding jump-shift instructions is a jump to the position of the instruction itself. We have among other things shown that the extraction of threads involved in that way corresponds to a finite-state execution mechanism that by making use of a counter can produce each finite-state thread from some program that is a finite or periodic infinite sequence of instructions from a finite set.

In the course of this work, we got convinced that a general format for the defining equations of thread extraction operations can be devised that yields thread extraction operations corresponding to execution mechanisms that can produce each finite-state thread from some program. One of the options for future work is to investigate this matter.

References

1. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.

2. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
3. J. A. Bergstra and C. A. Middelburg. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
4. J. A. Bergstra and C. A. Middelburg. Instruction sequences with dynamically instantiated instructions. Electronic Report PRG0710, Programming Research Group, University of Amsterdam, November 2007.
5. J. A. Bergstra and C. A. Middelburg. Instruction sequences with indirect jumps. Electronic Report PRG0709, Programming Research Group, University of Amsterdam, November 2007.
6. J. A. Bergstra and C. A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.
7. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
8. J. A. Bergstra and A. Ponse. Interface groups for analytic execution architectures. Electronic Report PRG0601, Programming Research Group, University of Amsterdam, May 2006.
9. W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin, 2000.
10. A. Ponse and M. B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al., editors, *CiE 2006*, volume 3988 of *Lecture Notes in Computer Science*, pages 445–458. Springer-Verlag, 2006.
11. D. Sannella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 13–30. Springer-Verlag, Berlin, 1999.
12. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, Amsterdam, 1990.

Within this series the following reports appeared.

[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/