On the behaviours produced by instruction sequences under execution

Bergstra, J.A.; Middelburg, C.A.

Publication date
2011

Document Version
Final published version

Link to publication

Citation for published version (APA):
Bergstra, J. A., & Middelburg, C. A. (2011). *On the behaviours produced by instruction sequences under execution*. ArXiv. http://arxiv.org/abs/1106.6196

On the Behaviours Produced by Instruction Sequences under Execution

J.A. Bergstra and C.A. Middelburg

Informatics Institute, Faculty of Science, University of Amsterdam,
Science Park 904, 1098 XH Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. We study several aspects of the behaviours produced by instruction sequences under execution in the setting of the algebraic theory of processes known as ACP. We use ACP to describe the behaviours produced by instruction sequences under execution and to describe two protocols implementing these behaviours in the case where the processing of instructions takes place remotely. We also show that all finite-state behaviours considered in ACP can be produced by instruction sequences under execution.

Keywords: instruction sequence, remote instruction processing, instruction sequence producible process

1 Introduction

The concept of an instruction sequence is a very primitive concept in computing. It has always played a central role in computing because of the fact that execution of instruction sequences underlies virtually all past and current generations of computers. It happens that, given a precise definition of an appropriate notion of an instruction sequence, many issues in computer science can be clearly explained in terms of instruction sequences. A simple yet interesting example is that a program can simply be defined as a text that denotes an instruction sequence. Such a definition corresponds to an empirical perspective found among practitioners.

In theoretical computer science, the meaning of programs usually plays a prominent part in the explanation of many issues concerning programs. Moreover, what is taken for the meaning of programs is mathematical by nature. On the other hand, it is customary that practitioners do not fall back on the mathematical meaning of programs in case explanation of issues concerning programs is needed. They phrase their explanations from an empirical perspective. An empirical perspective that we consider appealing is the perspective that a program is in essence an instruction sequence and an instruction sequence under execution produces a behaviour that is controlled by its execution environment in the sense that each step performed actuates the processing of an instruction by the execution environment and a reply returned at completion of the processing determines how the behaviour proceeds.

This paper concerns the behaviours produced by instruction sequences under execution as such and two issues relating to the behaviours produced by instruction sequences under execution, namely the issue of implementing these behaviours in the case where the processing of instructions takes place remotely and the issue of the extent to which the behaviours considered in process algebra can be produced by instruction sequences under execution.

Remote instruction processing means that a stream of instructions to be processed arises at one place and the processing of that stream of instructions is handled at another place. This phenomenon is increasingly encountered. It is found if loading the instruction sequence to be executed as a whole is impracticable. For instance, the storage capacity of the execution unit is too small or the execution unit is too far away. Remote instruction processing requires special attention because the transmission time of the messages involved in remote instruction processing makes it hard to keep the execution unit busy without intermission.

In the literature on computer architecture, hardly anything can be found that contributes to a sound understanding of the phenomenon of remote instruction processing. As a first step towards such an understanding, we give rigorous descriptions of two protocols for remote instruction processing at a level of abstraction that captures the underlying essence of the protocols. One protocol is very simple, but makes no effort keep the execution unit busy without intermission. The other protocol is more complex and is directed towards keeping the execution unit busy without intermission. It is reminiscent of an instruction pre-fetching mechanism as found in pipelined processors (see e.g. [26]), but its range of application is not restricted to pipelined instruction processing.

The work presented in this paper belongs to a line of research which started with an attempt to approach the semantics of programming languages from the perspective mentioned above. The first published paper on this approach is [7]. That paper is superseded by [8] with regard to the groundwork for the approach: program algebra, an algebraic theory of single-pass instruction sequences, and basic thread algebra, an algebraic theory of mathematical objects that represent in a direct way the behaviours produced by instruction sequences under execution.[1] The main advantages of the approach are that it does not require a lot of mathematical background and that it is more appealing to practitioners than the main approaches to programming language semantics: the operational approach, the denotational approach and the axiomatic approach. For an overview of these approaches, see e.g. [32].

The work presented in this paper is based on the instruction sequences considered in program algebra and the representation of the behaviours produced by instruction sequences under execution considered in basic thread algebra. It is rather awkward to describe and analyse the behaviours of this kind using al-

[1] In [8], basic thread algebra is introduced under the name basic polarized process algebra.

gebraic theories of processes such as ACP [3,6], CCS [27,31] and CSP [23,29]. However, the objects considered in basic thread algebra can be viewed as representations of processes as considered in process algebra. This allows for the protocols for remote instruction processing to be described using ACP or rather ACP$^\tau$, an extension of ACP which supports abstraction from internal actions.

Process algebra is an area of the study of concurrency which is considered relevant to computer science, as is witnesses by the extent of the work on algebraic theories of processes such as ACP, CCS and CSP in theoretical computer science. This strongly hints that there are programmed systems whose behaviours can be taken for processes as considered in process algebra. Therefore, it is interesting to know to which extent the behaviours considered in process algebra can be produced by programs under execution, starting from the perception of a program as an instruction sequence. In this paper, we will show that, by apposite choice of instructions, all finite-state processes can be produced by instruction sequences (provided that the cluster fair abstraction rule, see e.g. Section 5.6 of [24], is valid).

The instruction sequences considered in program algebra are single-pass instruction sequences, i.e. finite or infinite sequences of instructions of which each instruction is executed at most once and can be dropped after it has been executed or jumped over. Program algebra does not provide a notation for programs that is intended for actual programming: programs written in an assembly language are finite instruction sequences for which single-pass execution is usually not possible. We will also show that all finite-state processes can as well be produced by programs written in a program notation which is close to existing assembly languages.

Instruction sequences under execution may make use of services provided by their execution environment such as counters, stacks and Turing tapes. The use operators added to basic thread algebra in e.g. [12] can be used to describe the behaviours produced by instruction sequences under execution that make use of services. Interesting is that instruction sequences under execution that make use of services may produce infinite-state processes. On that account, we will make precise what processes are produced by instruction sequences under execution that make use of services provided by their execution environment.

As a continuation of the work on a new approach to programming language semantics mentioned above, the notion of an instruction sequence was subjected to systematic and precise analysis using the groundwork laid earlier. This led among other things to expressiveness results about the instruction sequences considered and variations of the instruction sequences considered (see e.g. [12,18,20,21,36]). Instruction sequences are under discussion for many years in diverse work on computer architecture, as witnessed by e.g. [4,22,25,30,33,34,35,39,41], but the notion of an instruction sequence has never been subjected to any precise analysis before. As another continuation of the work on a new approach to programming language semantics mentioned above, selected issues relating to well-known subjects from the theory of computation and the area of computer architecture were rigorously investigated thinking in terms of instruction sequences (see e.g. [14,15,16,17,19]). The subjects from the theory of computation, namely the halting problem and non-uniform computational complexity, are usually investigated thinking in terms of a common model of computation such as Turing machines and Boolean circuits (see e.g. [1,28,38]). The subjects from the area of computer architecture, namely instruction sequence performance, instruction set architectures and remote instruction processing, are usually not investigated in a rigorous way at all. The general aim of the work in both continuations mentioned is to bring instruction sequences as a theme in computer science better into the picture. The work presented in this paper forms a part of the last mentioned continuation.

This paper is organized as follows. The body of the paper consists of three parts. The first part (Sections 2–7) concerns the behaviours produced by instruction sequences under execution as such and includes surveys of program algebra, basic thread algebra and the algebraic theory of processes known as ACP. The second part (Sections 8–10) concerns the issue of implementing these behaviours in the case where the processing of instructions takes place remotely and includes rigorous descriptions of two protocols for remote instruction processing. The third part (Sections 11–14) concerns the issue of the extent to which the behaviours considered in process algebra can be produced by instruction sequences under execution and includes the result that, by apposite choice of instructions, all finite-state processes can be produced by instruction sequences.

This paper consolidates material from [11,13,14].

2 Program Algebra

In this section, we review PGA (ProGram Algebra). The starting-point of program algebra is the perception of a program as a single-pass instruction sequence. The concepts underlying the primitives of program algebra are common in programming, but the particular form of the primitives is not common. The predominant concern in the design of program algebra has been to achieve simple syntax and semantics, while maintaining the expressive power of arbitrary finite control.

In PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of *basic instructions* has been given. The intuition is that the execution of a basic instruction may modify a state and produces a reply at its completion. The possible replies are the Boolean values T and F.

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. On execution of an instruction sequence, these primitive instructions have the following effects:

– the effect of a positive test instruction $+a$ is that 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 — if there is no primitive instructions to proceed with, inaction occurs;
– the effect of a negative test instruction $-a$ is the same as the effect of $+a$, but with the role of the value produced reversed;
– the effect of a plain basic instruction a is the same as the effect of $+a$, but 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 there is no primitive instructions to proceed with, inaction 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 is a countably infinite set of variables which includes x, y, z. Terms are built as usual. We use infix notation for concatenation and postfix notation for repetition.

A closed PGA term is considered to denote a non-empty, finite or eventually periodic infinite sequence of primitive instructions.[2] The instruction sequence denoted by a closed term of the form $t \,;\, t'$ is the instruction sequence denoted by t concatenated with the instruction sequence denoted by t'. The instruction sequence denoted by a closed term of the form t^{ω} is the instruction sequence denoted by t concatenated infinitely many times with itself. Some simple examples of closed PGA terms are

$$a \,;\, b \,;\, c \;, \qquad +a \,;\, \#2 \,;\, \#3 \,;\, b \,;\, ! \;, \qquad a \,;\, (b \,;\, c)^{\omega} \;.$$

On execution of the instruction sequence denoted by the first term, the basic instructions a, b and c are executed in that order and after that inaction occurs. On execution of the instruction sequence denoted by the second term, the basic instruction a is executed first, if the execution of a produces the reply T, the basic instruction b is executed next and after that execution terminates, and if the execution of a produces the reply F, inaction occurs. On execution of the instruction sequence denoted by the third term, the basic instruction a is executed first, and after that the basic instructions b and c are executed in that order repeatedly forever.

Closed PGA terms are considered equal if they represent the same instruction sequence. The axioms for instruction sequence equivalence are given in Table 1. In this table, n stands for an arbitrary positive natural number. The term t^n,

[2] An eventually periodic infinite sequence is an infinite sequence with only finitely many distinct suffixes.

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

where t is a PGA term, is defined by induction on n as follows: $t^1 = t$ and $t^{n+1} = t \,;\, t^n$. The *unfolding* equation $x^\omega = x \,;\, x^\omega$ is derived as follows:

$$\begin{aligned} x^\omega &= (x \,;\, x)^\omega && \text{by PGA2} \\ &= x \,;\, (x \,;\, x)^\omega && \text{by PGA4} \\ &= x \,;\, x^\omega && \text{by PGA2}\,. \end{aligned}$$

Each closed PGA term is derivably equal to a term in *canonical form*, i.e. a term of the form t or $t \,;\, t'^\omega$, where t and t' are closed PGA terms in which the repetition operator does not occur. For example:

$$\begin{aligned} &(a \,;\, b)^\omega \,;\, c \,;\, ! = a \,;\, (b \,;\, a)^\omega\;, \\ &{+a} \,;\, (\#4 \,;\, b \,;\, (-c \,;\, \#5 \,;\, !)^\omega)^\omega = {+a} \,;\, \#4 \,;\, b \,;\, (-c \,;\, \#5 \,;\, !)^\omega\;. \end{aligned}$$

The initial models of PGA are considered its standard models. Henceforth, we restrict ourselves to the initial model $\boldsymbol{\mathcal{I}}_{\mathrm{PGA}}$ of PGA in which:

- the domain is the set of all non-empty, finite and eventually periodic infinite sequences over the set $\mathfrak{I}$ of primitive instructions;
- the operation associated with $\,;\,$ is concatenation;
- the operation associated with $^\omega$ is the operation $\underline{\omega}$ defined as follows:
  - if F is a finite sequence over $\mathfrak{I}$, then $F^{\underline{\omega}}$ is the unique eventually periodic infinite sequence F' such that F concatenated n times with itself is a proper prefix of F' for each $n \in \mathbb{N}$;
  - if F is an eventually periodic infinite sequence over $\mathfrak{I}$, then $F^{\underline{\omega}}$ is F.

In the sequel, we use the term *instruction sequence* for the elements of the domain of $\boldsymbol{\mathcal{I}}_{\mathrm{PGA}}$, and we denote the interpretations of the constants and operators of PGA in $\boldsymbol{\mathcal{I}}_{\mathrm{PGA}}$ by the constants and operators themselves. $\boldsymbol{\mathcal{I}}_{\mathrm{PGA}}$ is loosely called *the* initial model of PGA because all initial models of PGA are isomorphic, i.e. there exist bijective homomorphism between them (see e.g. [37,40]).

3 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra). BTA is an algebraic theory of mathematical objects that represent in a direct way the behaviours produced by instruction sequences under execution. The objects concerned 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

Table 2. Axiom of BTA

$x \unlhd \mathsf{tau} \unrhd y = x \unlhd \mathsf{tau} \unrhd x$	T1

$\mathcal{A} \cup \{\mathsf{tau}\}$. A thread performs basic actions in a sequential fashion. Upon each basic action performed, a reply from an execution environment determines how it proceeds. The possible replies are the Boolean values 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 13.

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 *inaction* 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 $\_ \unlhd a \unrhd \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

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 operators. We introduce *basic action prefixing* as an abbreviation: $a \circ t$, where $a \in \mathcal{A}_{\mathsf{tau}}$ and t is a term of sort $\mathbf{T}$, abbreviates $t \unlhd a \unrhd t$.

The thread denoted by a closed term of the form $t \unlhd a \unrhd t'$ will first perform a, and then proceed as the thread denoted by t if the reply from the execution environment is T and proceed as the thread denoted by t' if the reply from the execution environment is F. The threads denoted by D and S will become inactive and terminate, respectively. Some simple examples of closed BTA terms are

$$a \circ (\mathsf{S} \unlhd b \unrhd \mathsf{D}) \;, \qquad (b \circ \mathsf{S}) \unlhd a \unrhd \mathsf{D} \;.$$

The first term denotes the thread that first performs basic action a, next performs basic action b, if the reply from the execution environment on performing b is T, after that terminates, and if the reply from the execution environment on performing b is F, after that becomes inactive. The second term denotes the thread that first performs basic action a, if the reply from the execution environment on performing a is T, next performs the basic action b and after that terminates, and if the reply from the execution environment on performing a is F, next becomes inactive.

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

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 $t \unlhd a \unrhd t'$ with t and t' that contain only variables

Table 3. RDP, RSP and AIP

$\langle X \mid E\rangle = \langle t_X \mid E\rangle$ if $X = t_X \in E$	RDP	$\pi_0(x) = \mathsf{D}$	P0
$E \Rightarrow X = \langle X \mid E\rangle$ if $X \in \mathrm{V}(E)$	RSP	$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1
		$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\bigwedge_{n\geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	AIP	$\pi_{n+1}(x \unlhd a \unrhd y) = \pi_n(x) \unlhd a \unrhd \pi_n(y)$	P3

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 [5].

A simple example of a guarded recursive specification is the one consisting of following two equations:

$$x = x \unlhd a \unrhd y \;, \qquad y = y \unlhd b \unrhd \mathsf{S} \;.$$

The x-component of the solution of this guarded recursive specification is the thread that first performs basic action a repeatedly until the reply from the execution environment on performing a is F, next performs basic action b repeatedly until the reply from the execution environment on performing b is F, and after that terminates.

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X \mid E\rangle$ of sort $\mathbf{T}$ standing for the X-component of the unique solution of E. We write $\langle t_X \mid E\rangle$ for t_X with, for all $Y \in \mathrm{V}(E)$, all occurrences of Y in t_X replaced by $\langle Y \mid E\rangle$. The axioms for the constants for the components of the solutions of guarded recursive specifications are RDP (Recursive Definition Principle) and RSP (Recursive Specification Principle), which are given in Table 3. RDP and RSP are actually axiom schemas in which X stands for an arbitrary variable, t_X stands for an arbitrary BTA term, and E stands for an arbitrary guarded recursive specification over BTA. Side conditions are added to restrict what X, t_X and E stand for. The equations $\langle X \mid E\rangle = \langle t_X \mid E\rangle$ for a fixed E express that the constants $\langle X \mid E\rangle$ make up a solution of E. The conditional equations $E \Rightarrow X = \langle X \mid E\rangle$ express that this solution is the only one.

RDP and RSP are means to prove closed terms that denote the same infinite thread equal. We introduce AIP (Approximation Induction Principle) as an additional means to prove closed terms that denote the same infinite thread equal. 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. AIP is also given in Table 3. Here, approximation up to depth n is phrased in terms of the unary *projection* operator $\pi_n : \mathbf{T} \to \mathbf{T}$. The axioms for the projection operators are axioms P0–P3 in Table 3. P1–P3 are actually axiom schemas in which a stands for arbitrary basic action and n stands for an arbitrary natural number.

We write BTA+REC for BTA extended with the constants for the components of the solutions of guarded recursive specifications, the projection operators and the axioms RDP, RSP, AIP and P0–P3.

The minimal models of BTA+REC are considered its standard models.[3] Recall that a model of an algebraic theory is minimal iff all elements of the domains associated with the sorts of the theory can be denoted by closed terms. Henceforth, we restrict ourselves to the minimal models of BTA+REC. We assume that a minimal model $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$ of BTA+REC has been given.

In the sequel, we use the term *thread* for the elements of the domain of $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$, and we denote the interpretations of constants and operators in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$ by the constants and operators themselves.

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

- $T \in \mathit{Res}(T)$;
- if $T' \trianglelefteq a \trianglerighteq T'' \in \mathit{Res}(T)$, then $T' \in \mathit{Res}(T)$ and $T'' \in \mathit{Res}(T)$.

Let T be a thread and let $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$. Then T is *regular over* $\mathcal{A}'$ if the following conditions are satisfied:

- $\mathit{Res}(T)$ is finite;
- for all $T', T'' \in \mathit{Res}(T)$ and $a \in \mathcal{A}_{\mathsf{tau}}$, $T' \trianglelefteq a \trianglerighteq T'' \in \mathit{Res}(T)$ implies $a \in \mathcal{A}'$.

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

For example, the x-component of the solution of the guarded recursive specification consisting of the following two equations:

$$x = a \circ y \;, \qquad y = (c \circ y) \trianglelefteq b \trianglerighteq (x \trianglelefteq d \trianglerighteq \mathsf{S})$$

has five states and is regular over any $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$ for which $\{a, b, c, d\} \subseteq \mathcal{A}'$.

We will make use of the fact that being a regular thread coincides with being a component of 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 BTA is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$ over BTA, where each t_X is a term of the form D, S or $Y \trianglelefteq a \trianglerighteq Z$ with $Y, Z \in V$.

Proposition 1. *Let T be a thread and let $\mathcal{A}' \subseteq \mathcal{A}_{\mathsf{tau}}$. Then T is regular over $\mathcal{A}'$ iff there exists a finite linear recursive specification E over* BTA *in which only basic actions from $\mathcal{A}'$ occur such that T is a component of the solution of E.*

Proof. The implication from left to right is proved as follows. Because T is regular, $\mathit{Res}(T)$ is finite. Hence, there are finitely many threads $T_1, \ldots, T_n$, with $T = T_1$, such that $\mathit{Res}(T) = \{T_1, \ldots, T_n\}$. Now T is the x_1-component of the solution of the linear recursive specification consisting of the following equations:

$$x_i = \begin{cases} \mathsf{S} & \text{if } T_i = \mathsf{S} \\ \mathsf{D} & \text{if } T_i = \mathsf{D} \\ x_j \trianglelefteq a \trianglerighteq x_k & \text{if } T_i = T_j \trianglelefteq a \trianglerighteq T_k \end{cases} \qquad \text{for all } i \in [1, n] \;.$$

[3] A minimal model of an algebraic theory is a model of which no proper subalgebra is a model as well.

Table 4. Defining equations for thread extraction operation

$\lvert a\rvert = a \circ \mathsf{D}$	$\lvert\#l\rvert = \mathsf{D}$
$\lvert a\,;F\rvert = a \circ \lvert F\rvert$	$\lvert\#0\,;F\rvert = \mathsf{D}$
$\lvert{+a}\rvert = a \circ \mathsf{D}$	$\lvert\#1\,;F\rvert = \lvert F\rvert$
$\lvert{+a}\,;F\rvert = \lvert F\rvert \unlhd a \unrhd \lvert\#2\,;F\rvert$	$\lvert\#l+2\,;u\rvert = \mathsf{D}$
$\lvert{-a}\rvert = a \circ \mathsf{D}$	$\lvert\#l+2\,;u\,;F\rvert = \lvert\#l+1\,;F\rvert$
$\lvert{-a}\,;F\rvert = \lvert\#2\,;F\rvert \unlhd a \unrhd \lvert F\rvert$	$\lvert!\rvert = \mathsf{S}$
	$\lvert!\,;F\rvert = \mathsf{S}$

Because T is regular over $\mathcal{A}'$, only basic actions from $\mathcal{A}'$ occur in the linear recursive specification constructed in this way.

The implication from right to left is proved as follows. Thread T is a component of the unique solution of a finite linear specification in which only basic actions from $\mathcal{A}'$ occur. This means that there are finitely many threads $T_1, \ldots,$ T_n, with $T = T_1$, such that for every $i \in [1,n]$, $T_i = \mathsf{S}$, $T_i = \mathsf{D}$ or $T_i = T_j \unlhd a \unrhd T_k$ for some $j,k \in [1,n]$ and $a \in \mathcal{A}'$. Consequently, $T' \in \mathit{Res}(T)$ iff $T' = T_i$ for some $i \in [1,n]$ and moreover $T' \unlhd a \unrhd T'' \in \mathit{Res}(T)$ only if $a \in \mathcal{A}'$. Hence, $\mathit{Res}(T)$ is finite and T is regular over $\mathcal{A}'$. $\square$

Remark 1. A structural operational semantics of BTA+REC and a bisimulation equivalence based on it can be found in e.g. [10]. The quotient algebra of the algebra of closed terms of BTA+REC by this bisimulation equivalence is one of the minimal models of BTA+REC.

4 Thread Extraction

In this short section, we use BTA+REC to make mathematically precise which threads are produced by instruction sequences under execution.

For that purpose, $\mathcal{A}$ is taken such that $\mathcal{A} \supseteq \mathfrak{A}$ is satisfied.

The *thread extraction* operation $\lvert\_\rvert$ assigns a thread to each instruction sequence. The thread extraction operation is defined by the equations given in Table 4 (for $a \in \mathfrak{A}$, $l \in \mathbb{N}$, and $u \in \mathfrak{I}$) and the rule that $\lvert\#l\,;F\rvert = \mathsf{D}$ if $\#l$ is the beginning of an infinite jump chain. This rule is formalized in e.g. [12].

Let F be an instruction sequence and T be a thread. Then we say that F *produces* T if $\lvert F\rvert = T$. For example,

$$\begin{array}{lll}
a\,;b\,;c & \text{produces} & a \circ b \circ c \circ \mathsf{D}\;, \\
{+a}\,;\#2\,;\#3\,;b\,;! & \text{produces} & (b \circ \mathsf{S}) \unlhd a \unrhd \mathsf{D}\;, \\
{+a}\,;{-b}\,;c\,;! & \text{produces} & (\mathsf{S} \unlhd b \unrhd (c \circ \mathsf{S})) \unlhd a \unrhd (c \circ \mathsf{S})\;, \\
{+a}\,;\#2\,;(b\,;\#2\,;c\,;\#2)^\omega & \text{produces} & \mathsf{D} \unlhd a \unrhd (b \circ \mathsf{D})\;.
\end{array}$$

In the case of instruction sequences that are not finite, the produced threads can be described as components of the solution of a guarded recursive specification. For example, the infinite instruction sequence

$$(a\,;{+b})^\omega$$

produces the x-component of the solution of the guarded recursive specification consisting of following two equations:

$$x = a \circ y \; , \qquad y = x \unlhd b \unrhd y$$

and the infinite instruction sequence

$$a \,;\, (+b \,;\, \#2 \,;\, \#3 \,;\, c \,;\, \#4 \,;\, -d \,;\, ! \,;\, a)^\omega$$

produces the x-component of the solution of the guarded recursive specification consisting of following two equations:

$$x = a \circ y \; , \qquad y = (c \circ y) \unlhd b \unrhd (x \unlhd d \unrhd \mathsf{S}) \; .$$

5 Algebra of Communicating Processes

In this section, we review ACP$^\tau$ (Algebra of Communicating Processes with abstraction). This algebraic theory of processes will among other things be used to make precise what processes are produced by the threads denoted by closed terms of BTA+REC. For a comprehensive overview of ACP$^\tau$, the reader is referred to [3,24].

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

We assume that there are infinitely many variables, including x, y, z. Terms are built as usual. We use infix notation for the binary operators. The precedence conventions used with respect to the operators of ACP$^\tau$ are as follows: $+$ binds weaker than all others, $\cdot$ binds stronger than all others, and the remaining operators bind equally strong.

Let t and t' 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:

– the process denoted by e first performs atomic action e and next terminates successfully;
– the process denoted by τ performs an unobservable atomic action and next terminates successfully;
– the process denoted by δ can neither perform an atomic action nor terminate successfully;
– the process denoted by $t + t'$ behaves either as the process denoted by t or as the process denoted by t', but not both;
– the process denoted by $t \cdot t'$ first behaves as the process denoted by t and on successful termination of that process it next behaves as the process denoted by t';
– the process denoted by $t \parallel t'$ behaves as the process that proceeds with the processes denoted by t and t' in parallel;
– the process denoted by $t \mathbin{⌊\!⌊} t'$ behaves the same as the process denoted by $t \parallel t'$, except that it starts with performing an atomic action of the process denoted by t;
– the process denoted by $t \mid t'$ behaves the same as the process denoted by $t \parallel t'$, except that it starts with performing an atomic action of the process denoted by t and an atomic action of the process denoted by t' synchronously;
– the process denoted by $\partial_H(t)$ behaves the same as the process denoted by t, except that atomic actions from H are blocked;
– the process denoted by $\tau_I(t)$ behaves the same as the process denoted by t, 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 Table 5. CM2–CM3, CM5–CM7, C1–C4, D1–D4 and TI1–TI4 are actually axiom schemas in which a, b and c stand for arbitrary constants of ACP$^\tau$, and H and I stand for arbitrary subsets of A. ACP$^\tau$ is extended with guarded recursion like BTA.

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. We write $\mathrm{V}(E)$ for the set of all variables that occur in E. 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 are only interested models of ACP$^\tau$ in which guarded recursive specifications have unique solutions, such as the models of ACP$^\tau$ presented in [3].

For each guarded recursive specification E and each $X \in \mathrm{V}(E)$, we introduce a constant $\langle X|E\rangle$ standing for the X-component of the unique solution of E. We

Table 5. Axioms of ACP$^\tau$

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

Table 6. RDP, RSP and AIP

$\langle X\|E\rangle = \langle t_X\|E\rangle$ if $X = t_X \in E$	RDP	$\pi_0(a) = \delta$	PR1
$E \Rightarrow X = \langle X\|E\rangle$ if $X \in \mathrm{V}(E)$	RSP	$\pi_{n+1}(a) = a$	PR2
		$\pi_0(a \cdot x) = \delta$	PR3
$\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	AIP	$\pi_{n+1}(a \cdot x) = a \cdot \pi_n(x)$	PR4
		$\pi_n(x + y) = \pi_n(x) + \pi_n(y)$	PR5
		$\pi_n(\tau) = \tau$	PR6
		$\pi_n(\tau \cdot x) = \tau \cdot \pi_n(x)$	PR7

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 $\langle Y|E\rangle$. The axioms for the constants for the components of the solutions of guarded recursive specifications are RDP and RSP, which are given in Table 6. RDP and RSP are actually axiom schemas in which X stands for an arbitrary variable, t_X stands for an arbitrary ACP$^\tau$ term, and E stands for an arbitrary guarded recursive specification over ACP$^\tau$. Side conditions are added to restrict what X, t_X and E stand for.

Closed terms of ACP$^\tau$ extended with constants for the components of the solutions of guarded recursive specifications that denote the same process cannot always be proved equal by means of the axioms of ACP$^\tau$ together with RDP and RSP. We introduce AIP to remedy this. AIP is based on the view that two processes are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a process behaves the same as that process, except that it cannot perform any further atomic action after n atomic actions have been performed. AIP is given in Table 6. Here, approximation up to depth

n is phrased in terms of a unary *projection* operator π_n. The axioms for the projection operators are axioms PR1–PR7 in Table 6. PR1–PR7 are actually axiom schemas in which a stands for arbitrary constants of ACP$^\tau$ different from τ and n stands for an arbitrary natural number.

We write ACP$^\tau$+REC for ACP$^\tau$ extended with the constants for the components of the solutions of guarded recursive specifications, the projection operators, and the axioms RDP, RSP, AIP and PR1–PR7.

The minimal models of ACP$^\tau$+REC are considered its standard models. Henceforth, we restrict ourselves to the minimal models of ACP$^\tau$+REC. We assume that a fixed but arbitrary minimal model $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$ of ACP$^\tau$+REC has been given.

From Section 12, we will sometimes assume that CFAR (Cluster Fair Abstraction Rule) is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. CFAR says that a cluster of silent steps that has exits can be eliminated if all exits are reachable from everywhere in the cluster. A precise formulation of CFAR can be found in [24].

We use the term *process* for the elements from the domain of $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$, and we denote the interpretations of constants and operators in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$ 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 $e \cdot P' \in \mathit{Sub}(P)$, then $P' \in \mathit{Sub}(P)$;
– if $e \cdot P' + P'' \in \mathit{Sub}(P)$, then $P' \in \mathit{Sub}(P)$.

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 $P' \in \mathit{Sub}(P)$ and $e \in \mathsf{A}_\tau$, $e \cdot P' \in \mathit{Sub}(P)$ implies $e \in \mathsf{A}'$;
– for all $P', P'' \in \mathit{Sub}(P)$ and $e \in \mathsf{A}_\tau$, $e \cdot P' + P'' \in \mathit{Sub}(P)$ 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 a component of 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$^\tau$ is a guarded recursive specification $E = \{X = t_X \mid X \in V\}$ over ACP$^\tau$, where each t_X is linear.

Proposition 2. *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$^\tau$ *in which only atomic actions from A' occur such that P is a component of the solution of E.*

Proof. The proof follows the same line as the proof of Proposition 1. □

Remark 2. Proposition 2 is concerned with processes that are regular over A. We can also prove that being a regular process over A_τ coincides with being a component of the solution of a finite linear recursive specification over ACP$^\tau$if we assume that the cluster fair abstraction rule [24] holds in the model $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. However, we do not need this more general result.

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

6 Program-Service Interaction Instructions

Recall that, in PGA, 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 PGA in which the following additional assumptions relating to $\mathfrak{A}$ are made:

- a fixed but arbitrary finite set $\mathcal{F}$ of *foci* has been given;
- a fixed but arbitrary finite set $\mathcal{M}$ of *methods* has been given;
- $\mathfrak{A} = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.

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 of the form $f.m$ is taken as making a request to the service named f to process command m.

A basic instruction of the form $f.m$ is called a *program-service interaction instruction*. Recall that, in BTA, it is assumed that a fixed but arbitrary set $\mathcal{A}$ of basic actions has been given. In the sequel, we will make use of a version of BTA in which $\mathcal{A} = \mathfrak{A}$. A basic action of the form $f.m$ is called a *thread-service interaction action*.

The intuition concerning program-service interaction instructions given above will be made fully precise in Section 7, using ACP.

7 Process Extraction

In this section, we use ACP$^\tau$+REC to make mathematically precise which processes are produced by threads.

For that purpose, A and $|$ are taken such that the following conditions are satisfied:[4]

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

[4] As usual, we will write $\mathbb{B}$ for the set $\{\mathsf{T}, \mathsf{F}\}$.

Table 7. Defining equations for process extraction operation

$\lvert \mathsf{S} \rvert^{\mathrm{c}} = \mathrm{stop}$
$\lvert \mathsf{D} \rvert^{\mathrm{c}} = \delta$
$\lvert T \unlhd \mathsf{tau} \unrhd T' \rvert^{\mathrm{c}} = \mathrm{i} \cdot \mathrm{i} \cdot \lvert T \rvert^{\mathrm{c}}$
$\lvert T \unlhd f.m \unrhd T' \rvert^{\mathrm{c}} = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot \lvert T \rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot \lvert T' \rvert^{\mathrm{c}})$

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

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

Actions of the forms $\mathrm{s}_f(d)$ and $\mathrm{r}_f(d)$ are send and receive actions, respectively, stop is an explicit termination action, and i is a concrete internal action.

The *process extraction* operation $\lvert\_\rvert$ assigns a process to each thread. The process extraction operation $\lvert\_\rvert$ is defined by $\lvert T \rvert = \tau_{\{\mathrm{stop}\}}(\lvert T \rvert^{\mathrm{c}})$, where $\lvert\_\rvert^{\mathrm{c}}$ is defined by the equations given in Table 7 (for $f \in \mathcal{F}$ and $m \in \mathcal{M}$).

Let P be a process, T be a thread, and F be an instruction sequence. Then we say that T *produces* P if $\tau \cdot \tau_I(\lvert T \rvert) = \tau \cdot P$ for some $I \subseteq \mathsf{A}$, and we say that F *produces* P if $\lvert F \rvert$ produces P.

Notice that 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. Notice also that, for each thread T, $\lvert T \rvert^{\mathrm{c}}$ is a process that in the event of termination performs a special termination action just before termination. Abstraction from this termination action yields the process denoted by $\lvert T \rvert$.

The process extraction operation preserves the axioms of BTA+REC. Before we make this fully precise, we have a closer look at the axioms of BTA+REC.

A proper axiom is an equation or a conditional equation. In Table 3, we do not find proper axioms. Instead of proper axioms, we find axiom schemas without side conditions and axiom schemas with side conditions. The axioms of BTA+REC are obtained by replacing each axiom schema by all its instances.

Henceforth, we write α^*, where α is a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$, for the unique homomorphic extension of α to terms of BTA+REC. Moreover, we identify $t_1 = t_2$ and $\emptyset \Rightarrow t_1 = t_2$.

Proposition 3. *Let $E \Rightarrow t_1 = t_2$ be an axiom of* BTA+REC*, and let α be a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$. Then $\lvert \alpha^*(t_1) \rvert = \lvert \alpha^*(t_2) \rvert$ if $\lvert \alpha^*(t'_1) \rvert = \lvert \alpha^*(t'_2) \rvert$ for all $t'_1 = t'_2 \in E$.*

Proof. The proof is trivial for the axiom of BTA and the axioms RDP and RSP. Using the equation $\lvert \pi_n(T) \rvert^{\mathrm{c}} = \pi_{2n}(\lvert T \rvert^{\mathrm{c}})$, the proof is also trivial for the axioms AIP and P0–P3. This equation is easily proved by induction on n and case distinction on the structure of T in both the basis step and the inductive step. □

Remark 3. Proposition 3 would go through if no abstraction of the above-mentioned special termination action was made. Notice further that ACP$^\tau$ 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.

8 A Simple Protocol for Remote Instruction Processing

In this section and the next section, we consider two protocols for remote instruction processing. The simple protocol described in this section is presumably the most straightforward protocol for remote instruction processing that can be achieved. Therefore, we consider it a suitable starting-point for the design of more advanced protocols for remote instruction processing – such as the one described in the next section. Before this simple protocol is described, an extension of ACP is introduced to simplify the description of the protocols.

The following extension of ACP from [2] will be used: the non-branching conditional operator $:\rightarrow$ over $\mathbb{B}$. The expression $b :\rightarrow p$, is to be read as `if` b `then` p `else` δ. The additional axioms for the non-branching conditional operator are

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

In the sequel, we will use expressions whose evaluation yields Boolean values instead of the constants T and F. Because the evaluation of the expressions concerned are not dependent on the processes denoted by the terms in which they occur, we will identify each such expression with the constant for the Boolean value that its evaluation yields. Further justification of this can be found in [9, Section 9].

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

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

In the case where the thread is S or D, the instruction stream generator sends a special instruction (stop or dead) and the instruction stream execution unit does not send back a reply.

In this section, we consider a very simple protocol for remote instruction processing that makes no effort to keep the execution unit busy without intermission.

In the protocols, the generation of an instruction stream start from the thread produced by an instruction sequence under execution instead of the instruction sequence itself. It follows immediately from the definition of the thread extraction operation that the threads produced by instruction sequences under execution are regular threads. Therefore, we restrict ourselves to regular threads.

We write $\mathcal{I}$ for the set $\mathcal{A} \cup \{\mathsf{stop}, \mathsf{dead}\}$. Elements from $\mathcal{I}$ will loosely be called instructions. The restriction of the domain of $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$ to the regular threads will be denoted by $\mathcal{RT}$.

The functions *act*, *thrt*, and *thrf* defined below give, for each thread T different from S and D, the basic action that T will perform first, the thread with which it will proceed if the reply from the execution environment is T, and the thread with which it will proceed if the reply from the execution environment is F, respectively. The functions $\mathit{act}{:}\mathcal{RT} \to \mathcal{I}$, $\mathit{thrt}{:}\mathcal{RT} \to \mathcal{RT}$, and $\mathit{thrf}{:}\mathcal{RT} \to \mathcal{RT}$ are defined as follows:

$$\begin{array}{lll}
\mathit{act}(\mathsf{S}) = \mathsf{stop}\ , & \mathit{thrt}(\mathsf{S}) = \mathsf{D}\ , & \mathit{thrf}(\mathsf{S}) = \mathsf{D}\ , \\
\mathit{act}(\mathsf{D}) = \mathsf{dead}\ , & \mathit{thrt}(\mathsf{D}) = \mathsf{D}\ , & \mathit{thrf}(\mathsf{D}) = \mathsf{D}\ , \\
\mathit{act}(T \trianglelefteq a \trianglerighteq T') = a\ , & \mathit{thrt}(T \trianglelefteq a \trianglerighteq T') = T\ , & \mathit{thrf}(T \trianglelefteq a \trianglerighteq T') = T'\ .
\end{array}$$

The function nxt^0 defined below is used by the instruction stream generator to distinguish when it starts with handling the instruction to be executed next between the different instructions that it may be. The function $\mathit{nxt}^0{:}\mathcal{I} \times \mathcal{RT} \to \mathbb{B}$ is defined as follows:

$$\mathit{nxt}^0(a, T) = \begin{cases} \mathsf{T} & \text{if } \mathit{act}(T) = a \\ \mathsf{F} & \text{if } \mathit{act}(T) \neq a\ . \end{cases}$$

For the purpose of describing the simple protocol outlined above in ACP$^\tau$, A and $\mid$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, the following conditions are satisfied:

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

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

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

Notice that the set $\mathbb{B}$ is the set of replies.

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

$$\partial_H(\mathit{ISG}^0_T \parallel \mathit{IMTC}^0 \parallel \mathit{RTC}^0 \parallel \mathit{ISEU}^0)\ ,$$

where the process ISG^{0}_{T} is recursively specified by the following equation:

$$\begin{aligned}\mathit{ISG}^{0}_{T} = \sum_{f.m\in\mathcal{A}} & \mathit{nxt}^{0}(f.m,T) :\to \\ & \quad \mathrm{s}_1(f.m)\cdot(\mathrm{r}_4(\mathsf{T})\cdot \mathit{ISG}^{0}_{\mathit{thrt}(T)} + \mathrm{r}_4(\mathsf{F})\cdot \mathit{ISG}^{0}_{\mathit{thrf}(T)}) \\ & + \mathit{nxt}^{0}(\mathsf{stop},T) :\to \mathrm{s}_1(\mathsf{stop}) + \mathit{nxt}^{0}(\mathsf{dead},T) :\to \mathrm{s}_1(\mathsf{dead})\ ,\end{aligned}$$

the process IMTC^{0} is recursively specified by the following equation:

$$\mathit{IMTC}^{0} = \sum_{d\in\mathcal{I}} \mathrm{r}_1(d)\cdot \mathrm{s}_2(d)\cdot \mathit{IMTC}^{0}\ ,$$

the process RTC^{0} is recursively specified by the following equation:

$$\mathit{RTC}^{0} = \sum_{r\in\mathbb{B}} \mathrm{r}_3(r)\cdot \mathrm{s}_4(r)\cdot \mathit{RTC}^{0}\ ,$$

the process ISEU^{0} is recursively specified by the following equation:

$$\begin{aligned}\mathit{ISEU}^{0} = & \sum_{f.m\in\mathcal{A}} \mathrm{r}_2(f.m)\cdot \mathrm{s}_f(m)\cdot(\mathrm{r}_f(\mathsf{T})\cdot \mathrm{s}_3(\mathsf{T}) + \mathrm{r}_f(\mathsf{F})\cdot \mathrm{s}_3(\mathsf{F}))\cdot \mathit{ISEU}^{0} \\ & + \mathrm{r}_2(\mathsf{stop}) + \mathrm{r}_2(\mathsf{dead})\cdot\delta\end{aligned}$$

and

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

ISG^{0}_{T} is the instruction stream generator for thread T, IMTC^{0} is the transmission channel for messages containing instructions, RTC^{0} is the transmission channel for replies, and ISEU^{0} is the instruction stream execution unit.

If we abstract from all communications via the transmission channels, then the process denoted by $\partial_H(\mathit{ISG}^{0}_{T} \parallel \mathit{IMTC}^{0} \parallel \mathit{RTC}^{0} \parallel \mathit{ISEU}^{0})$ and the process $|T|$ are equal modulo an initial silent step.

Theorem 1. *For each $T\in\mathcal{RT}$, $\tau\cdot\tau_{\{\mathrm{j}\}}(\partial_H(\mathit{ISG}^{0}_{T} \parallel \mathit{IMTC}^{0} \parallel \mathit{RTC}^{0} \parallel \mathit{ISEU}^{0}))$ denotes the process $\tau\cdot|T|$.*

Proof. Let $T\in\mathcal{RT}$. Moreover, let E be a finite linear recursive specification over ACP$^{\tau}$ with $X\in \mathrm{V}(E)$ such that $|T|$ is the X-component of the solution of E in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^{\tau}+\mathrm{REC}}$. By Proposition 2 and the definition of the process extraction operation, it is sufficient to prove that

$$\tau\cdot\tau_{\{\mathrm{j}\}}(\partial_H(\mathit{ISG}^{0}_{T} \parallel \mathit{IMTC}^{0} \parallel \mathit{RTC}^{0} \parallel \mathit{ISEU}^{0})) = \tau\cdot\langle X|E\rangle\ .$$

By AIP, it is sufficient to prove that for all $n\geq 0$:

$$\pi_n(\tau\cdot\tau_{\{\mathrm{j}\}}(\partial_H(\mathit{ISG}^{0}_{T} \parallel \mathit{IMTC}^{0} \parallel \mathit{RTC}^{0} \parallel \mathit{ISEU}^{0}))) = \pi_n(\tau\cdot\langle X|E\rangle)\ .$$

This is easily proved by induction on n and in the inductive step by case distinction on the structure of T, using the axioms of ACP$^{\tau}$ and RDP and in addition the fact that $|T'|\in \mathit{Sub}(|T|)$ for all $T'\in \mathit{Res}(T)$ and the fact that there exists an bijection between $\mathit{Sub}(|T|)$ and $\mathrm{V}(E)$. □

9 A More Complex Protocol

In this section, we consider a more complex protocol for remote instruction processing that makes an effort to keep the execution unit busy without intermission.

The specifics of the more complex protocol considered here are that:

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

This protocol is reminiscent of an instruction pre-fetching mechanism as found in pipelined processors (see e.g. [26]), but its range of application is not restricted to pipelined instruction processing.

We write $\mathbb{B}^{\leq n}$, where $n \in \mathbb{N}$, for the set $\{u \in \mathbb{B}^* \mid \mathit{len}(u) \leq n\}$.[5]

It is assumed that a natural number ℓ has been given. The number ℓ is taken for the maximal number of steps that the instruction stream generator may run ahead of the instruction stream execution unit. Whether the execution unit can be kept busy without intermission with the given ℓ depends on the actual execution times of instructions and the actual transmission times over the transmission channels involved. If the execution unit can be kept busy without intermission with the given ℓ, then it is useless to increase ℓ.

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\mathit{updcr}(r,(n,R)) = (n+1, \{(u,T) \mid (ru,T) \in R\})\ .$$

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

$$\mathit{sel}(R) = \{(u,T) \in R \mid \forall (v,T') \in R \bullet \mathit{len}(u) \leq \mathit{len}(v)\}\ .$$

Notice that $(u,T) \in \mathit{sel}(R)$ and $(v,T') \in R$ only if $\mathit{len}(u) \leq \mathit{len}(v)$. By that breadth-first run-ahead is enforced. The performance of the protocol would change considerably if breadth-first run-ahead was not enforced.

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

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

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

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

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

$$\mathit{updcm}((k,u,a),(n,S)) = (n \mathbin{\dot{-}} k, S \cup \{(\mathit{tl}^{n \dot{-} k}(u), a)\})\ .^{6}$$

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

$$\mathit{updpr}(r, (n, S)) = (n + 1, \{(u, a) \mid (ru, a) \in S\}) \ .$$

The function nxt defined below is used by the instruction stream execution unit to distinguish when it starts with handling the instruction to be executed next between the different instructions that it may be. The function $\mathit{nxt} : \mathcal{I} \times \mathcal{P}(\mathbb{B}^{\leq \ell} \times \mathcal{I}) \to \mathbb{B}$ is defined as follows:

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

The instruction stream execution unit sends replies via a reply transmission channel to the instruction stream generator. We refer to a succession of transmitted replies as a *reply stream.*

For the purpose of describing the transmission protocol in ACP$^\tau$, A and $\mid$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, the following conditions are satisfied:

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

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

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

Let $T \in \mathcal{RT}$. Then the process representing the more complex protocol for remote instruction processing with regard to thread T is described by

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

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

$$\begin{aligned} \mathit{ISG}_T &= \mathit{ISG}'_{(0,\{(\epsilon,T)\})} \ , \\ \mathit{ISG}'_{(n,R)} &= \sum_{(u,T) \in \mathit{sel}(R)} \mathrm{s}_1((n, u, \mathit{act}(T))) \cdot \mathit{ISG}'_{\mathit{updpm}((u,T),(n,R))} \\ &\quad + \sum_{r \in \mathbb{B}} \mathrm{r}_4(r) \cdot \mathit{ISG}'_{\mathit{updcr}(r,(n,R))} \\ &\quad (\text{for every } (n, R) \in \mathcal{S}_{\text{ISG}} \text{ with } R \neq \emptyset) \ , \\ \mathit{ISG}'_{(n,\emptyset)} &= \mathrm{j} \\ &\quad (\text{for every } (n, \emptyset) \in \mathcal{S}_{\text{ISG}}) \ , \end{aligned}$$

[6] As usual, we write $i \mathbin{\dot{-}} j$ for the monus of i and j, i.e. $i \mathbin{\dot{-}} j = i - j$ if $i \geq j$ and $i \mathbin{\dot{-}} j = 0$ otherwise. As usual, $\mathit{tl}^n(u)$ is defined by induction on n as follows: $\mathit{tl}^0(u) = u$ and $\mathit{tl}^{n+1}(u) = \mathit{tl}(\mathit{tl}^n(u))$.

the process *IMTC* is recursively specified by the following equation:

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

the process *RTC* is recursively specified by the following equation:

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

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

$$\begin{aligned}
ISEU \quad &= ISEU'_{(0,\emptyset)} \;, \\
ISEU'_{(n,S)} \quad &= \sum_{d \in \mathcal{IM}} \mathrm{r}_2(d) \cdot ISEU'_{updcm(d,(n,S))} \\
&\quad + \sum_{f.m \in \mathcal{A}} nxt(f.m, S) :\rightarrow \mathrm{s}_f(m) \cdot ISEU''_{(f,(n,S))} \\
&\quad + nxt(\mathsf{stop}, S) :\rightarrow \mathsf{j} + nxt(\mathsf{dead}, S) :\rightarrow \delta \\
&\text{(for every } (n,S) \in \mathcal{S}_{\text{ISEU}}) \;, \\
ISEU''_{(f,(n,S))} &= \sum_{r \in \mathbb{B}} \mathrm{r}_f(r) \cdot \mathrm{s}_3(r) \cdot ISEU'_{updpr(r,(n,S))} \\
&\quad + \sum_{d \in \mathcal{IM}} \mathrm{r}_2(d) \cdot ISEU''_{(f,updcm(d,(n,S)))} \\
&\text{(for every } (f,(n,S)) \in \mathcal{F} \times \mathcal{S}_{\text{ISEU}}) \;,
\end{aligned}$$

and

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

ISG_T is the instruction stream generator for thread T, *IMTC* is the transmission channel for instruction messages, *RTC* is the transmission channel for replies, and *ISEU* is the instruction stream execution unit.

The protocol described above has been designed such that, for each $T \in \mathcal{RT}$, $\tau \cdot \tau_{\{\mathsf{j}\}}(\partial_H(ISG_T \parallel IMTC \parallel RTC \parallel ISEU))$ denotes the process $\tau \cdot |T|$. We refrain from presenting a proof of the claim that the protocol satisfies this because this paper is first and foremost a conceptual paper and the proof is straightforward but tedious.

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

Suppose that the transmission times over the transmission channels are small compared with the execution times of instructions. Even then the protocol described in Section 8 will always have to idle for a short time after the execution of an instruction, whereas after an initial phase the protocol described above will never have to idle after the execution of an instruction if the instruction stream generator may run a few steps ahead of the instruction stream execution unit.

10 Adaptations of the Protocol

In this section, we discuss some conceivable adaptations of the protocol described in Section 9. While we were thinking through the details of that protocol, various variations suggested themselves. The variations discussed below are among the most salient ones. We think they deserve mention. However, their discussion is not in depth. The reason for this is that these variations have not yet been investigated thoroughly.

Consider the case where, for each instruction, it is known what the probability is with which its execution leads to the reply T. This might give reason to adapt the protocol described in Section 9. Suppose that the instruction stream generator states do not only keep the sequences of replies after which threads must be performed, but also the sequences of instructions involved in producing those sequences of replies. Then the probability with which the sequences of replies will happen can be calculated and several conceivable adaptations of the protocol to this probabilistic knowledge are possible by mere changes in the selection of the sequence of replies and the instruction that will be part of the next instruction message produced by the instruction stream generator. Among those adaptations are:

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

At first sight, these adaptations are reminiscent of combinations of an instruction pre-fetching mechanism and a branch prediction mechanism as found in pipelined processors (see e.g. [26]). However, usually branch prediction mechanisms make use of statistics based on recently processed instructions instead of probabilistic knowledge of the kind used in the protocols sketched above.

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

11 Alternative Choice Instructions

Process algebra is an area of the study of concurrency which is considered relevant to computer science, as is witnesses by the extent of the work on algebraic theories of processes such as ACP, CCS and CSP in theoretical computer science. This strongly hints that there are programmed systems whose behaviours can be taken for processes as considered in process algebra. Therefore, it is interesting to know to which extent the behaviours considered in process algebra can be produced by programs under execution, starting from the perception of a program as an instruction sequence. In coming sections, we will establish results concerning the processes as considered in ACP that can be produced by instruction sequences under execution.

For the purpose of producing processes as considered in ACP, we need a version of PGA with special basic instructions to deal with the non-deterministic choice between alternatives that stems from the alternative composition of processes. Recall that, in PGA, it is assumed that a fixed but arbitrary set $\mathfrak{A}$ of basic instructions has been given. In the coming sections, we will make use a version of PGA in which the following additional assumptions relating to $\mathfrak{A}$ are made:

- a fixed but arbitrary finite set $\mathcal{F}$ of *foci* has been given;
- a fixed but arbitrary finite set $\mathcal{M}$ of *methods* has been given;
- a fixed but arbitrary set $\mathcal{AA}$ of *atomic actions*, with $\mathrm{t} \notin \mathcal{AA}$, has been given;
- $\mathfrak{A} = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\} \cup \{\mathsf{ac}(e_1, e_2) \mid e_1, e_2 \in \mathcal{AA} \cup \{\mathrm{t}\}\}$.

On execution of a basic instruction $\mathsf{ac}(e_1, e_2)$, first a non-deterministic choice between the atomic actions e_1 and e_2 is made and then the chosen atomic action is performed. The reply T is produced if e_1 is performed and the reply F is produced if e_2 is performed. Basic instructions of this kind are material to produce all regular processes by means of instruction sequences. A basic instruction of the form $\mathsf{ac}(e_1, e_2)$ is called an *alternative choice instruction*. Henceforth, we will write $\mathrm{PGA}_{\mathrm{ac}}$ for the version of PGA with alternative choice instructions.

The intuition concerning alternative choice instructions given above will be made fully precise at the end of this section, using ACP^{τ}. It will not be made fully precise using an extension of BTA because it is considered a basic property of threads that they are deterministic behaviours.

Recall that we make use of a version of BTA in which $\mathcal{A} = \mathfrak{A}$. A basic action of the form $\mathsf{ac}(e_1, e_2)$ is called an *alternative choice action*. Henceforth, we will write $\mathrm{BTA}^{\mathrm{ac}}$ for the version of BTA with alternative choice actions.

For the purpose of making precise what processes are produced by the threads denoted by closed terms of $\mathrm{BTA}^{\mathrm{ac}}$+REC, A and $\mid$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, the following conditions are satisfied:

$$\mathsf{A} \supseteq \mathcal{AA} \cup \{\mathrm{t}\}$$

Table 8. Additional defining equation for process extraction operation

$\lvert T \trianglelefteq \mathsf{ac}(e, e') \trianglerighteq T' \rvert^{\mathrm{c}} = e \cdot \lvert T \rvert^{\mathrm{c}} + e' \cdot \lvert T' \rvert^{\mathrm{c}}$

and for all $e, e' \in \mathsf{A}$:

$$e' \mid e = \delta \ \ \mathsf{if}\ e' \in \mathcal{AA} \cup \{\mathrm{t}\}\ .$$

The process extraction operation for BTA$^{\mathrm{ac}}$ has as defining equations the equations given in Table 7 and in addition the equation given in Table 8.

Proposition 3 goes through for BTA$^{\mathrm{ac}}$.

12 Instruction Sequence Producible Processes

It follows immediately from the definitions of the thread extraction and process extraction operations that the instruction sequences considered in PGA produce regular processes. The question is whether all regular processes are producible by these instruction sequences. In this section, we show that all regular processes can be produced by the instruction sequences with alternative choice instructions.

We will make use of the fact that all regular threads over $\mathcal{A}$ can be produced by the single-pass instruction sequences considered in PGA.

Proposition 4. *For each thread T that is regular over $\mathcal{A}$, there exists a* PGA *instruction sequence F such that F produces T, i.e. $|F| = T$.*

Proof. By Proposition 1, T is a component of the solution of some finite linear recursive specification E over BTA. There occur finitely many variables $X_0, \ldots, X_n$ in E. Assume that T is the X_0-component of the solution of E. Let F be the PGA instruction sequence $(F_0 \,;\, \ldots \,;\, F_n)^{\omega}$, where F_i is defined as follows ($0 \leq i \leq n$):

$$F_i = \begin{cases} ! \,;\, ! \,;\, ! & \mathsf{if}\ X_i = \mathsf{S} \in E \\ \#0 \,;\, \#0 \,;\, \#0 & \mathsf{if}\ X_i = \mathsf{D} \in E \\ +a \,;\, \#3{\cdot}(j{-}i){-}1 \,;\, \#3{\cdot}(k{-}i){-}2 & \mathsf{if}\ X_i = X_j \trianglelefteq a \trianglerighteq X_k \in E \wedge i < j \wedge i < k \\ +a \,;\, \#3{\cdot}(j{-}i){-}1 \,;\, \#3{\cdot}(n{+}1{-}(i{-}k)){-}2 & \mathsf{if}\ X_i = X_j \trianglelefteq a \trianglerighteq X_k \in E \wedge i < j \wedge i \geq k \\ +a \,;\, \#3{\cdot}(n{+}1{-}(i{-}j)){-}1 \,;\, \#3{\cdot}(k{-}i){-}2 & \mathsf{if}\ X_i = X_j \trianglelefteq a \trianglerighteq X_k \in E \wedge i \geq j \wedge i < k \\ +a \,;\, \#3{\cdot}(n{+}1{-}(i{-}j)){-}1 \,;\, \#3{\cdot}(n{+}1{-}(i{-}k)){-}2 & \mathsf{if}\ X_i = X_j \trianglelefteq a \trianglerighteq X_k \in E \wedge i \geq j \wedge i \geq k. \end{cases}$$

Then F is a PGA instruction sequence such that the interpretation of $|F| = T$. $\square$

All regular processes over $\mathcal{AA}$ can be produced by the instruction sequences considered in PGA$_{\mathrm{ac}}$.

Theorem 2. *Assume that CFAR is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^{\tau}+\mathrm{REC}}$. Then, for each process P that is regular over $\mathcal{AA}$, there exists an instruction sequence F in which only basic instructions of the form $\mathsf{ac}(e, \mathrm{t})$ occur such that F produces P, i.e. $\tau \cdot \tau_{\{\mathrm{t}\}}(||F||) = \tau \cdot P$.*

Proof. By Propositions 1, 2 and 4, it is sufficient to show that, for each finite linear recursive specification E over ACP$^\tau$ in which only atomic actions from $\mathcal{AA}$ occur, there exists a finite linear recursive specification E' over BTA$^{\mathrm{ac}}$ 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$^\tau$ 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 BTA$^{\mathrm{ac}}$ 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$^\tau$ that consists of the recursion equations

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

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 $X \in \mathrm{V}(E)$. $\square$

For example, assuming that CFAR is valid, the instruction sequence

$$\begin{aligned} &(+\mathsf{ac}(\mathrm{r}_3(\mathsf{T}),\mathrm{t})\ ;\#4\ ;+\mathsf{ac}(\mathrm{r}_3(\mathsf{F}),\mathrm{t})\ ;\#5\ ;\#7; \\ &\ +\mathsf{ac}(\mathrm{s}_4(\mathsf{T}),\mathrm{t})\ ;\#5\ ;\#9\ ;+\mathsf{ac}(\mathrm{s}_4(\mathsf{F}),\mathrm{t})\ ;\#2\ ;\#9)^\omega \end{aligned}$$

produces the reply transmission channel process RTC of which a guarded recursive specification is given in Section 9.

Remark 4. Theorem 2 with "$\tau\cdot\tau_{\{\mathrm{t}\}}(||F||) = \tau\cdot P$" replaced by "$||F|| = P$" can be established if PGA is extended with multiple-reply test instructions, see [11]. In that case, the assumption that CFAR is valid is superfluous.

13 Services and Use Operators

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. 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. 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.

A *service* H consists of

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

satisfying the following condition:

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

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

We introduce the sort $\mathbf{S}$ of *services*. However, we will not introduce constants and operators to build terms of this sort. The sort $\mathbf{S}$, standing for the set of all services, is considered a parameter of the extension of BTA being presented. Moreover, we introduce, for each $f \in \mathcal{F}$, the binary *use* operator $/_f : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. The axioms for these operators are given in Table 9. Intuitively, $T /_f H$ is the thread that results from processing all actions performed by thread T that are of the form $f.m$ by service H. When a basic action of the form $f.m$ performed by thread T is processed by service H, it is turned into the basic action tau and postconditional composition is removed in favour of basic action prefixing on the basis of the reply value produced.

We add the use operators to $\mathrm{PGA}_{\mathrm{ac}}$ as well. We will only use the extension in combination with the thread extraction operation $|\_|$ and define $|F /_f H| =$

Table 9. Axioms for use operators

$\mathsf{S} /_f H = \mathsf{S}$		U1
$\mathsf{D} /_f H = \mathsf{D}$		U2
$(x \trianglelefteq \mathsf{tau} \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq \mathsf{tau} \trianglerighteq (y /_f H)$		U3
$(x \trianglelefteq g.m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g.m \trianglerighteq (y /_f H)$	if $f \neq g$	U4
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{T}$	U5
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (y /_f \frac{\partial}{\partial m} H)$	if $H(m) = \mathsf{F}$	U6
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ \mathsf{D}$	if $H(m) = \mathsf{B}$	U7
$(x \trianglelefteq \mathsf{ac}(e_1, e_2) \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq \mathsf{ac}(e_1, e_2) \trianglerighteq (y /_f H)$		U8
$\pi_n(x /_f H) = \pi_n(\pi_n(x) /_f H)$		U9

$|F| /_f H$. Hence, $|F /_f H|$ denotes the thread produced by F if F 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.

In order to extend the process extraction operation to the use operators, we need an extension of ACP$^\tau$ with action renaming operators ρ_h, where $h{:}\mathsf{A}_\tau \to \mathsf{A}_\tau$ such that $h(\tau) = \tau$. The axioms for action renaming are given in [24]. Intuitively, $\rho_h(P)$ behaves as P with each atomic action replaced according to h. We write $\rho_{e' \mapsto e''}$ for the renaming operator ρ_h with h defined by $h(e') = e''$ and $h(e) = e$ if $e \neq e'$.

For the purpose of extending the process extraction operation to the use operators, A and $|$ are taken such that, in addition to the conditions mentioned at the beginning of Section 7, with everywhere $\mathbb{B}$ replaced by $\mathbb{B} \cup \{\mathsf{B}\}$, and the conditions mentioned at the end of Section 11, the following conditions are satisfied:

$$\mathsf{A} \supseteq \{\mathrm{s}_{\mathrm{serv}}(r) \mid r \in \mathbb{B} \cup \{\mathsf{B}\}\} \cup \{\mathrm{r}_{\mathrm{serv}}(m) \mid m \in \mathcal{M}\} \cup \{\mathrm{stop}^*\}$$

and for all $e \in \mathsf{A}$, $m \in \mathcal{M}$, and $r \in \mathbb{B} \cup \{\mathsf{B}\}$:

$$\begin{aligned} &\mathrm{s}_{\mathrm{serv}}(r) \mid e = \delta\ , && \mathrm{stop} \mid \mathrm{stop} = \mathrm{stop}^*\ , \\ &e \mid \mathrm{r}_{\mathrm{serv}}(m) = \delta\ , && \mathrm{stop}^* \mid e = \delta\ . \end{aligned}$$

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

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

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

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

To extend the process extraction operation to the use operators, the defining equation concerning the postconditional composition operators has to be adapted and a new defining equation concerning the use operators has to be

Table 10. Adapted and additional defining equations for process extraction operation

$\lvert T \unlhd f.m \unrhd T' \rvert^{\mathrm{c}} = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot \lvert T\rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot \lvert T'\rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{B}) \cdot \delta)$
$\lvert T \mathbin{/_f} H\rvert^{\mathrm{c}} = \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}\}}(\partial_{A_f}(\lvert T\rvert^{\mathrm{c}} \parallel \rho_{h_f}(\lvert H\rvert^{\mathrm{c}}))))$

added. These two equations are given in Table 10, where $\lvert H\rvert^{\mathrm{c}}$ is the X_H-component of the solution of

$$\{X_{H'} = \sum_{m \in \mathcal{M}} \mathrm{r}_{\mathrm{serv}}(m) \cdot \mathrm{s}_{\mathrm{serv}}(H'(m)) \cdot X_{\frac{\partial}{\partial m} H'} + \mathrm{stop} \mid H' \in \Delta(H)\}\ ,$$

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 9, the axioms for the use operators include proper axioms, which are all of the form $t_1 = t_2$, and axioms that have a semantic side condition, which are all of the form $t_1 = t_2$ if $H(m) = r$. By that, the precise formulation of the preservation result is somewhat complicated.

Proposition 5.

1. *Let $t_1 = t_2$ be a proper axiom for the use operators, and let α be a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$. Then $\lvert\alpha^*(t_1)\rvert = \lvert\alpha^*(t_2)\rvert$.*
2. *Let $t_1 = t_2$ if $H(m) = r$ be an axiom with semantic side condition for the use operators, and let α be a valuation of variables in $\boldsymbol{\mathcal{M}}_{\mathrm{BTA+REC}}$. Then $\lvert\alpha^*(t_1)\rvert = \lvert\alpha^*(t_2)\rvert$ if $H(m) = r$.*

Proof. The proof is straightforward. We sketch the proof for axiom U5. By the definition of the process extraction operation, it is sufficient to show that $\lvert(T \unlhd f.m \unrhd T') \mathbin{/_f} H\rvert^{\mathrm{c}} = \lvert\mathsf{tau} \circ (T \mathbin{/_f} \frac{\partial}{\partial m} H)\rvert^{\mathrm{c}}$ if $H(m) = \mathsf{T}$. In outline, this goes as follows:

$$\begin{aligned}
&\lvert(T \unlhd f.m \unrhd T') \mathbin{/_f} H\rvert^{\mathrm{c}} \\
&= \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}} \\
&\quad (\partial_{\{\mathrm{stop}\}}(\partial_{A_f}(\mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot \lvert T\rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{F}) \cdot \lvert T'\rvert^{\mathrm{c}} + \mathrm{r}_f(\mathsf{B}) \cdot \delta) \parallel \rho_{h_f}(\lvert H\rvert^{\mathrm{c}})))) \\
&= \mathrm{i} \cdot \mathrm{i} \cdot \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}\}}(\partial_{A_f}(\lvert T\rvert^{\mathrm{c}} \parallel \rho_{h_f}(\lvert\tfrac{\partial}{\partial m} H\rvert^{\mathrm{c}})))) \\
&= \lvert\mathsf{tau} \circ (T \mathbin{/_f} \tfrac{\partial}{\partial m} H)\rvert^{\mathrm{c}}\ .
\end{aligned}$$

In the first and third step, we apply defining equations of $\lvert\_\rvert^{\mathrm{c}}$. In the second step, we apply axioms of ACP$^\tau$+REC with action renaming, and use that $H(m) = \mathsf{T}$. □

Remark 5. Let F be a PGA$_{\mathrm{ac}}$ instruction sequence and H be a service. Then $\lvert\lvert F \mathbin{/_f} H\rvert\rvert$ is the process produced by F if F 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.

14 PGLD 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 in an alternative choice instruction. The latter result requires programs that make use of Boolean registers.

A hierarchy of program notations rooted in PGA is introduced in [8]. 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.

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

The effects of all instructions in common with PGA are as in PGA 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 inaction occurs. If l equals 0 or l is greater than the length of the program, then termination occurs.

We define the meaning of PGLD programs by means of a function `pgld2pga` from the set of all PGLD programs to the set of all closed PGA terms. This function is defined by

$$\texttt{pgld2pga}(u_1 ; \ldots ; u_k) = (\phi_1(u_1) ; \ldots ; \phi_k(u_k) ; ! ; !)^\omega ,$$

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

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

PGLD is as expressive as PGA. Before we make this fully precise, we introduce a useful notation.

Let α is a valuation of variables in $\mathcal{I}_{\mathrm{PGA}}$, and let α^* be the unique homomorphic extension of α to terms of PGA. Then $\alpha^*(t)$ is independent of α if t is a closed term, i.e. $\alpha^*(t)$ is uniquely determined by $\mathcal{I}_{\mathrm{PGA}}$. Therefore, we write $t^{\mathcal{I}_{\mathrm{PGA}}}$ for $\alpha^*(t)$ if t is a closed term.

Proposition 6. *For each closed* PGA *term t, there exists a* PGLD *program p such that* $|t^{\mathcal{I}_{\mathrm{PGA}}}| = |\texttt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}|$.

Proof. In [8], a number of functions (called embeddings in that paper) are defined, whose composition gives, for each closed PGA term t, a PGLD program p such that $|t^{\mathcal{I}_{\mathrm{PGA}}}| = |\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}|$. $\square$

Let p be a PGLD program and P be a process. Then we say that p *produces* P if $|\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}|$ produces P.

Below, we will write PGLD$_{\mathrm{ac}}$ for the version of PGLD in which the additional assumptions relating to $\mathfrak{A}$ mentioned in Section 11 are made. As a corollary of Theorem 2 and Proposition 6, we have that all regular processes over $\mathcal{AA}$ can be produced by PGLD$_{\mathrm{ac}}$ programs.

Corollary 1. *Assume that CFAR is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^{\tau}+\mathrm{REC}}$. Then, for each process P that is regular over $\mathcal{AA}$, there exists a* PGLD$_{\mathrm{ac}}$ *program p such that p produces P.*

We switch to the use of Boolean registers now. 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 $s \in \mathbb{B} \cup \{\mathsf{B}\}$. Then the *Boolean register service* with initial state s, written BR_s, is the service $(\mathbb{B} \cup \{\mathsf{B}\}, \mathit{eff}, \mathit{eff}, s)$, where the function eff is defined as follows ($b \in \mathbb{B}$):

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

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

Let p be a PGLD program and P be a process. Then we say that *p produces P using Boolean registers* if $(\ldots(|\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}| \,/_{\mathsf{br}:1}\, BR_{\mathsf{F}}) \ldots /_{\mathsf{br}:k}\, BR_{\mathsf{F}})$ produces P for some $k \in \mathbb{N}^{+}$.

We have that PGLD$_{\mathrm{ac}}$ programs in which no atomic action from $\mathcal{AA}$ occurs more than once in an alternative choice instruction can produce all regular processes over $\mathcal{AA}$ using Boolean registers.

Theorem 3. *Assume that CFAR is valid in $\boldsymbol{\mathcal{M}}_{\mathrm{ACP}^\tau+\mathrm{REC}}$. Then, for each process P that is regular over $\mathcal{AA}$, there exists a* $\mathrm{PGLD}_{\mathrm{ac}}$ *program p in which each atomic action from $\mathcal{AA}$ occurs no more than once in an alternative choice instruction such that p produces P using Boolean registers.*

Proof. By the proof of Theorem 2 given in Section 12, it is sufficient to show that, for each thread T that is regular over $\mathcal{A}$, there exist a PGLD program p in which each basic action from $\mathcal{A}$ occurs no more than once and a $k \in \mathbb{N}^+$ such that $(\ldots(|\mathtt{pgld2pga}(p)^{\mathcal{I}_{\mathrm{PGA}}}| /_{\mathsf{br}:1}\, BR_{\mathsf{F}}) \ldots /_{\mathsf{br}:k}\, BR_{\mathsf{F}}) = T$.

Let T be a thread that is regular over $\mathcal{A}$. We may assume that T is produced by a PGLD program p' of the following form:

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

where, for each $i \in [1,n]$, $k_i, k'_i \in [0, n-1]$ (cf. the proof of Proposition 2 from [36]). It is easy to see that the PGLD 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'. □

15 Conclusions

Using the algebraic theory of processes known as ACP, we have described two protocols to deal with the phenomenon that, on execution of an instruction sequence, a stream of instructions to be processed arises at one place and the processing of that stream of instructions is handled at another place. The more complex protocol is directed towards keeping the execution unit busy. In this way, we have brought the phenomenon better into the picture and have ascribed a sense to the term instruction stream which makes clear that an instruction stream is dynamic by nature, in contradistinction with an instruction sequence. We have also discussed some conceivable adaptations of the more complex protocol.

The description of the protocols start from the behaviours produced by instruction sequences under execution. By that we abstract from the instruction sequences which produce those behaviours. How instruction streams can be generated efficiently from instruction sequences is a matter that obviously requires investigations at a less abstract level. The investigations in question are an option for future work.

We believe that the more complex protocol described in this paper provides a setting in which basic techniques aimed at increasing processor performance, such as pre-fetching and branch prediction, can be studied at a more abstract level than usual (cf. [26]). In particular, we think that the protocol can serve as a starting-point for the development of a model with which trade-offs encountered in the design of processor architectures can be clarified. We consider investigations into this matter an interesting option for future work.

The fact that process algebra is an area of the study of concurrency which is considered relevant to computer science, strongly hints that there are 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 an instruction sequence. We have shown that, by apposite choice of basic instructions, all regular processes can be produced by means of instruction sequences as considered in PGA.

We have also made precise what processes are produced by instruction sequences under execution that make use of services. The reason for this is that 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. Arora, S., Barak, B.: Computational Complexity: A Modern Approach. Cambridge University Press, Cambridge (2009)
2. Baeten, J.C.M., Bergstra, J.A.: Process algebra with signals and conditions. In: Broy, M. (ed.) Programming and Mathematical Methods. NATO ASI Series, vol. F88, pp. 273–323. Springer-Verlag (1992)
3. Baeten, J.C.M., Weijland, W.P.: Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
4. Baker, H.G.: Precise instruction scheduling without a precise machine model. SIGARCH Computer Architecture News 19(6), 4–8 (1991)
5. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) Proceedings 30th ICALP. Lecture Notes in Computer Science, vol. 2719, pp. 1–21. Springer-Verlag (2003)
6. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1–3), 109–137 (1984)
7. Bergstra, J.A., Loots, M.E.: Program algebra for component code. Formal Aspects of Computing 12(1), 1–17 (2000)
8. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming 51(2), 125–156 (2002)
9. Bergstra, J.A., Middelburg, C.A.: Splitting bisimulations and retrospective conditions. Information and Computation 204(7), 1083–1138 (2006)
10. Bergstra, J.A., Middelburg, C.A.: Maurer computers with single-thread control. Fundamenta Informaticae 80(4), 333–362 (2007)
11. Bergstra, J.A., Middelburg, C.A.: Instruction sequences for the production of processes. `arXiv:0811.0436v2 [cs.PL]` (November 2008)
12. Bergstra, J.A., Middelburg, C.A.: Program algebra with a jump-shift instruction. Journal of Applied Logic 6(4), 553–563 (2008)

13. Bergstra, J.A., Middelburg, C.A.: A protocol for instruction stream processing. `arXiv:0905.2257v1 [cs.PL]` (May 2009)
14. Bergstra, J.A., Middelburg, C.A.: Transmission protocols for instruction streams. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. Lecture Notes in Computer Science, vol. 5684, pp. 127–139. Springer-Verlag (2009)
15. Bergstra, J.A., Middelburg, C.A.: Instruction sequences and non-uniform complexity theory. `arXiv:0809.0352v3 [cs.CC]` (July 2010)
16. Bergstra, J.A., Middelburg, C.A.: On the operating unit size of load/store architectures. Mathematical Structures in Computer Science 20(3), 395–417 (2010)
17. Bergstra, J.A., Middelburg, C.A.: Indirect jumps improve instruction sequence performance. `arXiv:0909.2089v2 [cs.PL]` (December 2011)
18. Bergstra, J.A., Middelburg, C.A.: Thread extraction for polyadic instruction sequences. Scientific Annals of Computer Science 21(2), 283–310 (2011)
19. Bergstra, J.A., Middelburg, C.A.: Instruction sequence processing operators. Acta Informatica 49(3), 139–172 (2012)
20. Bergstra, J.A., Middelburg, C.A.: On the expressiveness of single-pass instruction sequences. Theory of Computing Systems 50(2), 313–328 (2012)
21. Bergstra, J.A., Ponse, A.: An instruction sequence semigroup with involutive anti-automorphisms. Scientific Annals of Computer Science 19, 57–92 (2009)
22. Brock, C., Hunt, W.A.: Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In: ICCD '97. pp. 31–36 (1997)
23. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM 31(3), 560–599 (1984)
24. Fokkink, W.J.: Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, Berlin (2000)
25. Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., Gill, J.: MIPS: A microprocessor architecture. In: MICRO '82. pp. 17–22 (1982)
26. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco, third edn. (2003)
27. Hennessy, M., Milner, R.: Algebraic laws for non-determinism and concurrency. Journal of the ACM 32(1), 137–161 (1985)
28. Hermes, H.: Enumerability, Decidability, Computability. Springer-Verlag, Berlin (1965)
29. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
30. Lunde, A.: Empirical evaluation of some features of instruction set processor architectures. Communications of the ACM 20(3), 143–153 (1977)
31. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
32. Mosses, P.D.: Formal semantics of programming languages — an overview. Electronic Notes in Theoretical Computer Science 148, 41–73 (2006)
33. Nair, R., Hopkins, M.E.: Exploiting instruction level parallelism in processors by caching scheduled groups. SIGARCH Computer Architecture News 25(2), 13–25 (1997)
34. Ofelt, D., Hennessy, J.L.: Efficient performance prediction for modern microprocessors. In: SIGMETRICS '00. pp. 229–239 (2000)
35. Patterson, D.A., Ditzel, D.R.: The case for the reduced instruction set computer. SIGARCH Computer Architecture News 8(6), 25–33 (1980)
36. Ponse, A., van der Zwaag, M.B.: An introduction to program and thread algebra. In: Beckmann, A., et al. (eds.) CiE 2006. Lecture Notes in Computer Science, vol. 3988, pp. 445–458. Springer-Verlag (2006)

37. Sannella, D., Tarlecki, A.: Algebraic preliminaries. In: Astesiano, E., Kreowski, H.J., Krieg-Brückner, B. (eds.) Algebraic Foundations of Systems Specification, pp. 13–30. Springer-Verlag, Berlin (1999)
38. Sipser, M.: Introduction to the Theory of Computation. Thomson, Boston, MA, second edn. (2006)
39. Tennenhouse, D.L., Wetherall, D.J.: Towards an active network architecture. SIGCOMM Computer Communication Review 37(5), 81–94 (2007)
40. Wirsing, M.: Algebraic specification. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 675–788. Elsevier, Amsterdam (1990)
41. Xia, C., Torrellas, J.: Instruction prefetching of systems codes with layout optimized for reduced cache misses. In: ISCA '96. pp. 271–282 (1996)