University of Amsterdam
Programming Research Group

Thread Algebra for Sequential Poly-Threading

J.A. Bergstra
C.A. Middelburg

Report PRG0804 March 2008

J.A. Bergstra

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

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

C.A. Middelburg

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
The Netherlands

e-mail: kmiddelb@science.uva.nl

Programming Research Group Electronic Report Series

Thread Algebra for Sequential Poly-Threading*

J.A. Bergstra and C.A. Middelburg

Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
J.A.Bergstra@uva.nl,C.A.Middelburg@uva.nl

Abstract. Threads as considered in basic thread algebra are primarily looked upon as behaviours exhibited by sequential programs on execution. It is a fact of life that sequential programs are often fragmented. Consequently, fragmented program behaviours are frequently found. In this paper, we consider this phenomenon. We extend basic thread algebra with the barest mechanism for sequencing of threads that are taken for fragments. This mechanism, called sequential poly-threading, supports both autonomous and non-autonomous thread selection in sequencing. We relate the resulting theory to the algebraic theory of processes known as ACP and use it to describe analytic execution architectures suited for fragmented programs.

Keywords: thread algebra, sequential poly-threading, autonomous thread selection, non-autonomous thread selection, process algebra, analytic execution architecture.

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

1 Introduction

In [10], we considered fragmentation of sequential programs that take the form of instruction sequences in the setting of program algebra [4]. The objective of the current paper is to develop a theory of the behaviours exhibited by sequential programs on execution that covers the case where the programs have been split into fragments. It is a fact of life that sequential programs are often fragmented. We remark that an important reason for fragmentation of programs is that the execution architecture at hand to execute them sets bounds to the size of programs. However, there may also be other reasons for program fragmentation, for instance business economical reasons.

In [4], a start was made with a line of research in which sequential programs that take the form of instruction sequences and the behaviours exhibited by sequential programs on execution are investigated (see e.g. [3, 5, 15]). In this line of research, the view is taken that the behaviour exhibited by a sequential program

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

on execution takes the form of a thread as considered in basic thread algebra [4].[1] With the current paper, we carry on this line of research. Therefore, we consider program behaviour fragments that take the form of threads as considered in basic thread algebra.

We extend basic thread algebra with the barest mechanism for sequencing of threads that are taken for program behaviour fragments. This mechanism is called sequential poly-threading. Inherent in the behaviour exhibited by a program on execution is that it does certain steps only for the sake of getting reply values returned by some service provided by an execution environment and that way having itself affected by that service. In the setting of thread algebra, the use mechanism is introduced in [9] to allow for this kind of interaction. Sequential poly-threading supports the initialization of one of the services used every time a thread is started up. With sequential poly-threading, a thread selection is made whenever a thread ends up with the intent to achieve the start-up of another thread. That thread selection can be made in two ways: by the terminating thread or externally. We show how thread selections of the latter kind can be internalized.

Both threads and services look to be special cases of a more general notion of process. Therefore, it is interesting to know the connections of threads and services with processes as considered in theories about concurrent processes such as ACP [1], CCS [14] and CSP [13]. We show that threads and services as considered in the theory developed in this paper can be viewed as processes that are definable over the extension of ACP with conditions introduced in [6]. An analytic execution architecture is a model of a hypothetical execution environment for sequential programs that is designed for the purpose of explaining how a program may be executed. The notion of analytic execution architecture defined in [11] is suited to sequential programs that have not been split into fragments. We use the theory developed in this paper to describe analytic execution architectures suited to sequential programs that have been split into fragments.

The line of research carried on in this paper has two main themes: the theme of instruction sequences and the theme of threads. Both [10] and the current paper are concerned with program fragmentation, but [10] elaborates on the theme of instruction sequences and the current paper elaborates on the theme of threads. It happens that there are aspects of program fragmentation that can be dealt with at the level of instruction sequences, but cannot be dealt with at the level of threads. In particular, the ability to replace special instructions in an instruction sequence fragment by different ordinary instructions every time execution is switched over to that fragment cannot be dealt with at the level of threads. Threads, which are intended for explaining the meaning of sequential programs, turn out to be too abstract to deal with program fragmentation in full.

[1] In [4], basic thread algebra is introduced under the name basic polarized process algebra. Prompted by the development of thread algebra [9], which is a design on top of it, basic polarized process algebra has been renamed to basic thread algebra.

This paper is organized as follows. First, we review basic thread algebra and the use mechanism (Sections 2 and 3). Next, we extend basic thread algebra with sequential poly-threading and show how external thread selections in sequential poly-threading can be internalized (Sections 4 and 5). Following this, we review ACP with conditions and relate the theory developed in this paper with ACP with conditions (Sections 6 and 7). After that, we discuss analytic execution architectures suited for programs that have been fragmented (Section 8). Finally, we make some concluding remarks (Section 9).

2 Basic Thread Algebra

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

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

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

Although BTA is one-sorted, we make this sort explicit. The reason for this is that we will extend BTA with additional sorts in Sections 3 and 4.

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

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

Terms of sort $\mathbf{T}$ are built as usual (see e.g. [16, 17]). Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z.

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

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

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

Table 1. Axiom of BTA

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

Table 2. Axioms for guarded recursion

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

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

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables of sort $\mathbf{T}$ and each t_X is a term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' BTA terms of sort $\mathbf{T}$ that contain only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [2]. A thread that is the solution of a finite guarded recursive specification over BTA is called a *finite-state* thread.

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

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

3 Interaction of Threads with Services

A thread may perform certain basic actions only for the sake of having itself affected by some service. When processing a basic action performed by a thread, a service affects that thread by returning a reply value to the thread at completion of the processing of the basic action. In this section, we introduce the use

mechanism, which is concerned with this kind of interaction between threads and services.[2]

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

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

$$\forall \alpha \in \mathcal{M}^+, m \in \mathcal{M} \bullet (H(\alpha) = \mathsf{B} \Rightarrow H(\alpha \frown \langle m \rangle) = \mathsf{B}) \ .^3$$

We write $\mathcal{S}$ for the set of all services and $\mathcal{R}$ for the set $\{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$. Given a service H and a method $m \in \mathcal{M}$, the *derived service* of H after processing m, written $\frac{\partial}{\partial m} H$, is defined by $\frac{\partial}{\partial m} H(\alpha) = H(\langle m \rangle \frown \alpha)$.

A service H can be understood as follows:

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

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

The axioms for the use operators are given in Table 3. In this table, f and g stand for arbitrary foci from $\mathcal{F}$ and m stands for an arbitrary method from $\mathcal{M}$. Axioms TSU3 and TSU4 express that the action tau and basic actions of the form $g.m$ with $f \neq g$ are not processed. Axioms TSU5 and TSU6 express that a thread is affected by a service as described above when a basic action of the form $f.m$ performed by the thread is processed by the service. Axiom TSU7 expresses that deadlock takes place when a basic action to be processed is not accepted.

[2] This version of the use mechanism was first introduced in [9]. In later papers, it is also called thread-service composition.

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

Table 3. Axioms for use operators

$\mathsf{S} /_f H = \mathsf{S}$		TSU1
$\mathsf{D} /_f H = \mathsf{D}$		TSU2
$\mathsf{tau} \circ x /_f H = \mathsf{tau} \circ (x /_f H)$		TSU3
$(x \trianglelefteq g.m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g.m \trianglerighteq (y /_f H)$	if $\neg\ f = g$	TSU4
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{T}$	TSU5
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{tau} \circ (y /_f \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{F}$	TSU6
$(x \trianglelefteq f.m \trianglerighteq y) /_f H = \mathsf{D}$	if $H(\langle m \rangle) = \mathsf{B}$	TSU7

Let T stand for either BTA or BTA+REC. Then we will write T+TSU for T, taking the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$ for $\mathcal{A}$, extended with the use operators and the axioms from Table 3.

4 Sequential Poly-Threading

BTA is a theory of the behaviours exhibited by sequential programs on execution. To cover the case where the programs have been split into fragments, we extend BTA in this section with the barest mechanism for sequencing of threads that are taken for fragments. The resulting theory is called $\mathrm{TA}_{\mathrm{spt}}$.

Our general view on the way of achieving a joint behaviour of the program fragments in a collection of program fragments between which execution can be switched is as follows:

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

In order to obtain such a joint behaviour from the behaviours of the program fragments on execution, a mechanism is needed by which the start-up of another program fragment behaviour is effectuated whenever a program fragment behaviour ends up with the intent to achieve such a start-up. In the setting of BTA, taking threads for program fragment behaviours, this requires the introduction of an additional sort, additional constants and additional operators. In doing so it is supposed that a collection of threads that corresponds to a collection of program fragments between which execution can be switched takes the form of a sequence, called a thread vector.

Like in BTA+TSU, it is assumed that there is a fixed but arbitrary finite set $\mathcal{F}$ of foci and a fixed but arbitrary finite set $\mathcal{M}$ of methods. It is also assumed that $\mathsf{tls} \in \mathcal{F}$ and $\mathsf{init} \in \mathcal{M}$. The focus tls and the method init play special roles: tls

is the focus of a service that is initialized each time a thread is started up by the mechanism referred to above and init is the initialization method of that service. For the set $\mathcal{A}$ of basic actions, we take again the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.

$\mathrm{TA}_{\mathrm{spt}}$ has the sort $\mathbf{T}$ of BTA and in addition the sort $\mathbf{TV}$ of *thread vectors*. To build terms of sort $\mathbf{T}$, $\mathrm{TA}_{\mathrm{spt}}$ has the constants and operators of BTA and in addition the following additional constants and operators:

- for each $i \in \mathbb{N}$, the *internally controlled switch-over* constant $\mathsf{S}i : \mathbf{T}$;
- the *externally controlled switch-over* constant $\mathsf{E} : \mathbf{T}$;
- the binary *sequential poly-threading* operator $\boxplus : \mathbf{T} \times \mathbf{TV} \to \mathbf{T}$;
- for each $k \in \mathbb{N}^+$,[4] the k-ary *external choice* operator $\Box_k : \underbrace{\mathbf{T} \times \cdots \times \mathbf{T}}_{k \text{ times}} \to \mathbf{T}$.

To build terms of sort $\mathbf{TV}$, $\mathrm{TA}_{\mathrm{spt}}$ has the following constants and operators:

- the *empty thread vector* constant $\langle\,\rangle : \mathbf{TV}$;
- the unary *singleton thread vector* operator $\langle \_ \rangle : \mathbf{T} \to \mathbf{TV}$;
- the binary *thread vector concatenation* operator $\_ \frown \_ : \mathbf{TV} \times \mathbf{TV} \to \mathbf{TV}$.

Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{TV}$, including α.

In the context of the sequential poly-threading operator $\boxplus$, the constants $\mathsf{S}i$ and E are alternatives for the constant S which produce additional effects. Let $p, p_1, \ldots, p_n$ be closed terms of sort $\mathbf{T}$. Then $\boxplus(p, \langle p_1 \rangle \frown \ldots \frown \langle p_n \rangle)$ first behaves as p, but when p terminates:

- in the case where p terminates with S, it terminates;
- in the case where p terminates with $\mathsf{S}i$:
  - it continues by behaving as $\boxplus(p_i, \langle p_1 \rangle \frown \ldots \frown \langle p_n \rangle)$ if $1 \leq i \leq n$,
  - it deadlocks otherwise;
- in the case where p terminates with E, it continues by behaving as one of $\boxplus(p_1, \langle p_1 \rangle \frown \ldots \frown \langle p_n \rangle), \ldots, \boxplus(p_n, \langle p_1 \rangle \frown \ldots \frown \langle p_n \rangle)$ or it deadlocks.

Moreover, the basic action $\mathsf{tls.init}$ is performed between termination and continuation. In the case where p terminates with E, the choice between the alternatives is made externally. Nothing is stipulated about the effect that the constants $\mathsf{S}i$ and E produce in the case where they occur outside the context of the sequential poly-threading operator.

The sequential poly-threading operator concerns sequencing of threads. A thread selection involved in sequencing of threads is called an *autonomous thread selection* if the selection is made by the terminating thread. Otherwise, it is called a *non-autonomous thread selection*. The constants $\mathsf{S}i$ are meant for autonomous thread selections and the constant E is meant for non-autonomous thread selections. We remark that non-autonomous thread selections are immaterial to the joint behaviours of program fragments referred to above.

In the case of a non-autonomous thread selection, it comes to an external choice between a number of threads. The external choice operator $\Box_k$ concerns

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

Table 4. Axiom for sequential poly-threading operator

$\between(\mathsf{S},\alpha) = \mathsf{S}$	SPT1
$\between(\mathsf{D},\alpha) = \mathsf{D}$	SPT2
$\between(x \trianglelefteq a \trianglerighteq y,\alpha) = \between(x,\alpha) \trianglelefteq a \trianglerighteq \between(y,\alpha)$	SPT3
$\between(\mathsf{S}i,\langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) = \mathsf{tls.init} \circ \between(x_i,\langle x_1\rangle \frown \ldots \frown \langle x_n\rangle)$ if $1 \leq i \leq n$	SPT4
$\between(\mathsf{S}i,\langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) = \mathsf{D}$ if $i = 0 \vee i > n$	SPT5
$\between(\mathsf{E},\langle x_1\rangle \frown \ldots \frown \langle x_{n+1}\rangle) =$	SPT6
$\quad \boxdot_{n+1}(\mathsf{tls.init} \circ \between(x_1,\langle x_1\rangle \frown \ldots \frown \langle x_{n+1}\rangle),\ldots,\mathsf{tls.init} \circ \between(x_{n+1},\langle x_1\rangle \frown \ldots \frown \langle x_{n+1}\rangle))$	
$\between(\mathsf{E},\langle\,\rangle) = \mathsf{D}$	SPT7

external choice between k threads. Let p_1, ..., p_k be closed terms of sort $\mathbf{T}$. Then $\boxdot_k(p_1,\ldots,p_k)$ behaves as the outcome of an external choice between p_1, ..., p_k and D.

$\mathrm{TA}_{\mathrm{spt}}$ has the axioms of BTA and in addition the axioms given in Table 4. In this table, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. The additional axioms express that threads are sequenced by sequential poly-threading as described above. There are no axioms for the external choice operators because their basic properties cannot be expressed as equations or conditional equations. For each $k \in \mathbb{N}^+$, the basic properties of $\boxdot_k$ are expressed by the following disjunction of equations: $\bigvee_{i\in[1,k]} \boxdot_k(x_1,\ldots,x_k) = x_i \vee \boxdot_k(x_1,\ldots,x_k) = \mathsf{D}$.[5]

To be fully precise, we should give axioms concerning the constants and operators to build terms of the sort **TV** as well. We refrain from doing so because the constants and operators concerned are the usual ones for sequences.

Guarded recursion can be added to $\mathrm{TA}_{\mathrm{spt}}$ as it is added to BTA in Section 2. We will write $\mathrm{TA}_{\mathrm{spt}}$+REC for $\mathrm{TA}_{\mathrm{spt}}$ extended with the constants for solutions of guarded recursive specifications and axioms RDP and RSP.

The use mechanism can be added to $\mathrm{TA}_{\mathrm{spt}}$ as it is added to BTA in Section 3. Let T stand for either $\mathrm{TA}_{\mathrm{spt}}$ or $\mathrm{TA}_{\mathrm{spt}}$+REC. Then we will write T+TSU for T extended with the use operators and the axioms from Table 3.

5 Internalization of Non-Autonomous Thread Selection

In the case of non-autonomous thread selection, the selection of a thread is made externally. In this section, we show how non-autonomous thread selection can be internalized. For that purpose, we first extend $\mathrm{TA}_{\mathrm{spt}}$ with postconditional switching. Postconditional switching is like postconditional composition, but covers the case where services processing basic actions produce reply values from the set $\mathbb{N}$ instead of reply values from the set $\{\mathsf{T},\mathsf{F}\}$. Postconditional switching is convenient when internalizing non-autonomous thread selection, but it is not necessary.

[5] We use the notation $[n,m]$, where $n,m \in \mathbb{N}$, for the set $\{i \in \mathbb{N} \mid n \leq i \leq m\}$.

Table 5. Axioms for postconditional switching operators

$\mathsf{tau} \trianglerighteq_k (x_1, \ldots, x_k) = \mathsf{tau} \trianglerighteq_k (x_1, \ldots, x_1)$	
$\boxplus(a \trianglerighteq_k (x_1, \ldots, x_k), \alpha) = a \trianglerighteq_k (\boxplus(x_1, \alpha), \ldots, \boxplus(x_k, \alpha))$	
$\mathsf{tau} \trianglerighteq_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{tau} \trianglerighteq_k (x_1 \mathbin{/_f} H, \ldots, x_k \mathbin{/_f} H)$	
$g.m \trianglerighteq_k (x_1, \ldots, x_k) \mathbin{/_f} H = g.m \trianglerighteq_k (x_1 \mathbin{/_f} H, \ldots, x_k \mathbin{/_f} H)$	if $\neg\ f = g$
$f.m \trianglerighteq_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{tau} \circ (x_i \mathbin{/_f} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = i \wedge i \in [1, k]$
$f.m \trianglerighteq_k (x_1, \ldots, x_k) \mathbin{/_f} H = \mathsf{D}$	if $\neg\ H(\langle m \rangle) \in [1, k]$

For each $a \in \mathcal{A}_{\mathsf{tau}}$ and $k \in \mathbb{N}^+$, we introduce the k-ary *postconditional switch* operator $a \trianglerighteq_k : \underbrace{\mathbf{T} \times \cdots \times \mathbf{T}}_{k \text{ times}} \to \mathbf{T}$. Let $p_1, \ldots, p_k$ be closed terms of sort $\mathbf{T}$. Then $a \trianglerighteq_k (p_1, \ldots, p_k)$ will first perform action a, and then proceed as p_1 if the processing of a leads to the reply 1, $\ldots$, p_k if the processing of a leads to the reply k.

The axioms for the postconditional switching operators are given in Table 5. In this table, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$, f and g stand for arbitrary foci from $\mathcal{F}$, and m stands for an arbitrary method from $\mathcal{M}$.

We proceed with the internalization of non-autonomous thread selections. Let p, p_1, $\ldots$, p_k be closed terms of sort $\mathbf{T}$. The idea is that $\boxplus(p, \langle p_1 \rangle \frown \ldots \frown \langle p_k \rangle)$ can be internalized by:

- replacing in $\boxplus(p, \langle p_1 \rangle \frown \ldots \frown \langle p_k \rangle)$ all occurrences of E by $\mathsf{S}k{+}1$;
- appending a thread that can make the thread selections to the thread vector.

Simultaneous with the replacement of all occurrences of E by $\mathsf{S}k{+}1$, all occurrences of $\mathsf{S}k{+}1$ must be replaced by D to prevent inadvertent selections of the appended thread. When making a thread selection, the appended thread has to request the external environment to give the position of the thread that it would have selected itself. We make the simplifying assumption that the external environment can be viewed as a service.

Let p, p_1, $\ldots$, p_k be closed terms of sort $\mathbf{T}$. Then the *internalization* of $\boxplus(p, \langle p_1 \rangle \frown \ldots \frown \langle p_k \rangle)$ is

$$\boxplus(\rho(p), \langle \rho(p_1) \rangle \frown \ldots \frown \langle \rho(p_k) \rangle \frown \langle \mathrm{ext.sel} \trianglerighteq_k (\mathsf{S}1, \ldots, \mathsf{S}k) \rangle) \,,$$

where $\rho(p')$ is p' with simultaneously all occurrences of E replaced by $\mathsf{S}k{+}1$ and all occurrences of $\mathsf{S}k{+}1$ replaced by D. Here, it is assumed that $\mathrm{ext} \in \mathcal{F}$ and $\mathrm{sel} \in \mathcal{M}$.

Postconditional switching is not really necessary for internalization. Let $k_1 = \lfloor k/2 \rfloor$, $k_2 = \lfloor k_1/2 \rfloor$, $k_3 = \lfloor (k - k_1)/2 \rfloor$, $\ldots$. Using postconditional composition, first a selection can be made between $\{p_1, \ldots, p_{k_1}\}$ and $\{p_{k_1+1}, \ldots, p_k\}$, next a selection can be made between $\{p_1, \ldots, p_{k_2}\}$ and $\{p_{k_2+1}, \ldots, p_{k_1}\}$ or between $\{p_{k_1+1}, \ldots, p_{k_3}\}$ and $\{p_{k_3+1}, \ldots, p_k\}$, depending on the outcome of the previous

selection, etcetera. In this way, the number of actions performed to select a thread is between $\lfloor {}^2\log(k) \rfloor$ and $\lceil {}^2\log(k) \rceil$.

6 ACP with Conditions

In Section 7, we will investigate the connections of threads and services with the processes considered in ACP-style process algebras. We will focus on ACP$^{\mathrm{c}}$, an extension of ACP with conditions introduced in [6]. In this section, we shortly review ACP$^{\mathrm{c}}$.

ACP$^{\mathrm{c}}$ is an extension of ACP with conditional expressions in which the conditions are taken from a Boolean algebra. ACP$^{\mathrm{c}}$ has two sorts: (i) the sort $\mathbf{P}$ of *processes*, (ii) the sort $\mathbf{C}$ of *conditions*. In ACP$^{\mathrm{c}}$, it is assumed that the following has been given: a fixed but arbitrary set A (of actions), with $\delta \notin \mathsf{A}$, a fixed but arbitrary set C_{at} (of atomic conditions), and a fixed but arbitrary commutative and associative function $| : \mathsf{A} \cup \{\delta\} \times \mathsf{A} \cup \{\delta\} \to \mathsf{A} \cup \{\delta\}$ such that $\delta \mid a = \delta$ for all $a \in \mathsf{A} \cup \{\delta\}$. The function $|$ is regarded to give the result of synchronously performing any two actions for which this is possible, and to be δ otherwise. Henceforth, we write A_δ for $\mathsf{A} \cup \{\delta\}$.

Let p and q be closed terms of sort $\mathbf{P}$, ζ and ξ be closed term of sort $\mathbf{C}$, $a \in \mathsf{A}$, $H \subseteq \mathsf{A}$, and $\eta \in \mathsf{C}_{\mathsf{at}}$. Intuitively, the constants and operators to build terms of sort $\mathbf{P}$ that will be used to define the processes to which threads and services correspond can be explained as follows:

- δ can neither perform an action nor terminate successfully;
- a first performs action a unconditionally and then terminates successfully;
- $p + q$ behaves either as p or as q, but not both;
- $p \cdot q$ first behaves as p, but when p terminates successfully it continues as q;
- $\zeta :\rightarrow p$ behaves as p under condition ζ;
- $p \parallel q$ behaves as the process that proceeds with p and q in parallel;
- $\partial_H(p)$ behaves the same as p, except that actions from H are blocked.

Intuitively, the constants and operators to build terms of sort $\mathbf{C}$ that will be used to define the processes to which threads and services correspond can be explained as follows:

- η is an atomic condition;
- $\bot$ is a condition that never holds;
- $\top$ is a condition that always holds;
- $-\zeta$ is the opposite of ζ;
- $\zeta \sqcup \xi$ is either ζ or ξ;
- $\zeta \sqcap \xi$ is both ζ and ξ.

The remaining operators of ACP$^{\mathrm{c}}$ are of an auxiliary nature. They are needed to axiomatize ACP$^{\mathrm{c}}$. The axioms of ACP$^{\mathrm{c}}$ are given in [6].

We write $\sum_{i \in \mathcal{I}} p_i$, where $\mathcal{I} = \{i_1, \ldots, i_n\}$ and $p_{i_1}, \ldots, p_{i_n}$ are terms of sort $\mathbf{P}$, for $p_{i_1} + \ldots + p_{i_n}$. The convention is that $\sum_{i \in \mathcal{I}} p_i$ stands for δ if $\mathcal{I} = \emptyset$. We

use the notation $p \lhd \zeta \rhd q$, where p and q are terms of sort $\mathbf{P}$ and ζ is a term of sort $\mathbf{C}$, for $\zeta :\rightarrow p + -\zeta :\rightarrow q$.

A process is considered definable over $\mathrm{ACP}^{\mathrm{c}}$ if there exists a guarded recursive specification over $\mathrm{ACP}^{\mathrm{c}}$ that has that process as its solution.

A *recursive specification* over $\mathrm{ACP}^{\mathrm{c}}$ is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is a term of sort $\mathbf{P}$ that only contains variables from V. Let t be a term of sort $\mathbf{P}$ containing a variable X. Then an occurrence of X in t is *guarded* if t has a subterm of the form $a \cdot t'$ where $a \in \mathsf{A}$ and t' is a term containing this occurrence of X. Let E be a recursive specification over $\mathrm{ACP}^{\mathrm{c}}$. Then E is a *guarded recursive specification* if, in each equation $X = t_X \in E$, all occurrences of variables in t_X are guarded or t_X can be rewritten to such a term using the axioms of $\mathrm{ACP}^{\mathrm{c}}$ in either direction and/or the equations in E except the equation $X = t_X$ from left to right. We only consider models of $\mathrm{ACP}^{\mathrm{c}}$ in which guarded recursive specifications have unique solutions, such as the full splitting bisimulation models of $\mathrm{ACP}^{\mathrm{c}}$ presented in [6].

For each guarded recursive specification E and each variable X that occurs as the left-hand side of an equation in E, we introduce a constant of sort $\mathbf{P}$ standing for the unique solution of E for X. This constant is denoted by $\langle X|E\rangle$. The axioms for guarded recursion are also given in [6].

In order to express the use operators, we need an extension of $\mathrm{ACP}^{\mathrm{c}}$ with action renaming operators. Intuitively, the action renaming operator ρ_f, where $f : \mathsf{A} \to \mathsf{A}$, can be explained as follows: $\rho_f(p)$ behaves as p with each action replaced according to f. The axioms for action renaming are the ones given in [12] and in addition the equation $\rho_f(\phi :\rightarrow x) = \phi :\rightarrow \rho_f(x)$. We write $\rho_{a' \mapsto a''}$ for the renaming operator ρ_g with g defined by $g(a') = a''$ and $g(a) = a$ if $a \neq a'$.

In order to explain the connection of threads and services with $\mathrm{ACP}^{\mathrm{c}}$ fully, we need an extension of $\mathrm{ACP}^{\mathrm{c}}$ with the condition evaluation operators CE_h introduced in [6]. Intuitively, the condition evaluation operator CE_h, where h is a function on conditions that is preserved by $\bot$, $\top$, $-$, $\sqcup$ and $\sqcap$, can be explained as follows: $\mathsf{CE}_h(p)$ behaves as p with each condition replaced according to h. The important point is that, if $h(\zeta) \in \{\bot, \top\}$, all subterms of the form $\zeta :\rightarrow q$ can be eliminated. The axioms for condition evaluation are also given in [6].

7 Threads, Services and $\mathrm{ACP}^{\mathrm{c}}$-Definable Processes

In this section, we relate threads and services as considered in $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU to processes that are definable over $\mathrm{ACP}^{\mathrm{c}}$ with action renaming.

For that purpose, A, $|$ and C_{at} are taken as follows:

$$\begin{aligned}\mathsf{A} = {} & \{\mathrm{s}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathcal{R}\} \cup \{\mathrm{r}_f(d) \mid f \in \mathcal{F}, d \in \mathcal{M} \cup \mathcal{R}\} \\ & \cup \{\mathrm{s}_{\mathrm{ext}}(n) \mid n \in \mathbb{N}\} \cup \{\mathrm{r}_{\mathrm{ext}}(n) \mid n \in \mathbb{N}\} \cup \{\mathrm{stop}, \overline{\mathrm{stop}}, \mathrm{stop}^*, \mathrm{i}\} \\ & \cup \{\mathrm{s}_{\mathrm{serv}}(r) \mid r \in \mathcal{R}\} \cup \{\mathrm{r}_{\mathrm{serv}}(m) \mid m \in \mathcal{M}\}\ ;\end{aligned}$$

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

$$\begin{aligned}
&\mathrm{s}_f(d) \mid \mathrm{r}_f(d) = \mathrm{i} \;, && && \mathrm{stop} \mid \overline{\mathrm{stop}} = \mathrm{stop}^* \;, && \\
&\mathrm{s}_f(d) \mid a = \delta && \mathsf{if}\ a \neq \mathrm{r}_f(d) \;, && \mathrm{stop} \mid a = \delta && \mathsf{if}\ a \neq \overline{\mathrm{stop}} \;, \\
&a \mid \mathrm{r}_f(d) = \delta && \mathsf{if}\ a \neq \mathrm{s}_f(d) \;, && a \mid \overline{\mathrm{stop}} = \delta && \mathsf{if}\ a \neq \mathrm{stop} \;, \\
&\mathrm{s}_{\mathrm{ext}}(n) \mid \mathrm{r}_{\mathrm{ext}}(n) = \mathrm{i} \;, && && \mathrm{i} \mid a = \delta \;, && \\
&\mathrm{s}_{\mathrm{ext}}(n) \mid a = \delta && \mathsf{if}\ a \neq \mathrm{r}_{\mathrm{ext}}(n) \;, && \mathrm{s}_{\mathrm{serv}}(r) \mid a = \delta \;, && \\
&a \mid \mathrm{r}_{\mathrm{ext}}(n) = \delta && \mathsf{if}\ a \neq \mathrm{s}_{\mathrm{ext}}(n) \;, && a \mid \mathrm{r}_{\mathrm{serv}}(m) = \delta \;; &&
\end{aligned}$$

and

$$\mathsf{C}_{\mathsf{at}} = \{H(\langle m \rangle) = r \mid H \in \mathcal{S}, m \in \mathcal{M}, r \in \mathcal{R}\} \;.$$

For each $f \in \mathcal{F}$, the set $A_f \subseteq \mathsf{A}$ and the function $R_f : \mathsf{A} \to \mathsf{A}$ are defined as follows:

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

for all $a \in \mathsf{A}$, $m \in \mathcal{M}$ and $r \in \mathcal{R}$:

$$\begin{aligned}
&R_f(\mathrm{s}_{\mathrm{serv}}(r)) = \mathrm{s}_f(r) \;, \\
&R_f(\mathrm{r}_{\mathrm{serv}}(m)) = \mathrm{r}_f(m) \;, \\
&R_f(a) = a && \mathsf{if}\ \textstyle\bigwedge_{r' \in \mathcal{R}} a \neq \mathrm{s}_{\mathrm{serv}}(r') \wedge \bigwedge_{m' \in \mathcal{M}} a \neq \mathrm{r}_{\mathrm{serv}}(m') \;.
\end{aligned}$$

The sets A_f and the functions R_f are used below to express the use operators in terms of the operators of $\mathrm{ACP}^{\mathrm{c}}$ with action renaming.

For convenience, we introduce a special notation. Let α be a term of sort $\mathbf{TV}$, let $p_1, \ldots, p_n$ be terms of sort $\mathbf{T}$ such that $\alpha = \langle p_1 \rangle \curvearrowright \ldots \curvearrowright \langle p_n \rangle$, and let $i \in [1, n]$. Then we write $\alpha[i]$ for p_i.

We proceed with relating threads and services as considered in $\mathrm{TA}_{\mathrm{spt}}$+REC+ TSU to processes definable over $\mathrm{ACP}^{\mathrm{c}}$ with action renaming. The underlying idea is that threads and services can be viewed as processes that are definable over $\mathrm{ACP}^{\mathrm{c}}$ with action renaming. We define those processes by means of a translation function $[\![\_]\!]$ from the set of all terms of sort $\mathbf{T}$ to the set of all function from the set of all terms of sort $\mathbf{TV}$ to the set of all terms of sort $\mathbf{P}$ and a translation function $[\![\_]\!]$ from the set of all services to the set of all terms of sort $\mathbf{P}$. These translation functions are defined inductively by the equations given in Table 6, where we write in the last equation $t_{H'}$ for the term

$$\begin{aligned}
&\sum_{m \in \mathcal{M}} \mathrm{r}_{\mathrm{serv}}(m) \cdot \mathrm{s}_{\mathrm{serv}}(H'(\langle m \rangle)) \cdot (X_{\frac{\partial}{\partial m} H'} \lhd H'(\langle m \rangle) = \mathsf{T} \sqcup H'(\langle m \rangle) = \mathsf{F} \rhd X_{H'}) \\
&+ \overline{\mathrm{stop}} \;.
\end{aligned}$$

Let p be a closed term of sort $\mathbf{T}$. Then the *process algebraic interpretation* of p is $[\![p]\!](\langle\,\rangle)$. Henceforth, we write $[\![p]\!]$ for $[\![p]\!](\langle\,\rangle)$.

Table 6. Definition of translation functions

$[\![X]\!](\alpha) = X$	
$[\![\mathsf{S}]\!](\alpha) = \mathrm{stop}$	
$[\![\mathsf{D}]\!](\alpha) = \mathrm{i} \cdot \delta$	
$[\![t_1 \trianglelefteq \mathsf{tau} \trianglerighteq t_2]\!](\alpha) = \mathrm{i} \cdot \mathrm{i} \cdot [\![t_1]\!](\alpha)$	
$[\![t_1 \trianglelefteq f.m \trianglerighteq t_2]\!](\alpha) = \mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot [\![t_1]\!](\alpha) + \mathrm{r}_f(\mathsf{F}) \cdot [\![t_2]\!](\alpha))$	
$[\![\mathsf{S}i]\!](\alpha) = \mathsf{tls.init} \cdot [\![\alpha[i]]\!](\alpha)$	if $1 \leq i \leq \mathrm{len}(\alpha)$
$[\![\mathsf{S}i]\!](\alpha) = \mathrm{i} \cdot \delta$	if $i = 0 \vee i > \mathrm{len}(\alpha)$
$[\![\mathsf{E}]\!](\alpha) = \sum_{i \in [1,\mathrm{len}(\alpha)]} \mathrm{r}_{\mathrm{ext}}(i) \cdot \mathsf{tls.init} \cdot [\![\alpha[i]]\!](\alpha) + \mathrm{i} \cdot \delta$	
$[\![\boxplus(t, \alpha')]\!](\alpha) = [\![t]\!](\alpha')$	
$[\![\Box_k(t_1, \ldots, t_k)]\!](\alpha) = \sum_{i \in [1,k]} \mathrm{r}_{\mathrm{ext}}(i) \cdot [\![t_i]\!](\alpha) + \mathrm{i} \cdot \delta$	
$[\![\langle X \mid E \rangle]\!](\alpha) = \langle X \mid \{X = [\![t]\!](\alpha) \mid X = t \in E\} \rangle$	
$[\![t /_f H]\!](\alpha) = \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}, \overline{\mathrm{stop}}\}}(\partial_{A_f}([\![t]\!](\alpha) \between \rho_{R_f}([\![H]\!]))))$	
$[\![H]\!] = \langle X_H \mid \{X_{H'} = t_{H'} \mid H' \in \mathcal{S}\} \rangle$	

Notice that ACP is sufficient for the translation of terms of sort **T**: no conditional expressions occur in the translations. For the translation of services, we need the full power of $\mathrm{ACP}^{\mathrm{c}}$.

The translations given above preserve the axioms of $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU. Roughly speaking, this means that the translations of these axioms are derivable from the axioms of $\mathrm{ACP}^{\mathrm{c}}$ with action renaming and guarded recursion. Before we make this fully precise, we have a closer look at the axioms of $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU.

A proper axiom is an equation or a conditional equation. In Tables 1–4, we do not only find proper axioms. In addition to proper axioms, we find: (i) axiom schemas without side conditions; (ii) axiom schemas with syntactic side conditions; (iii) axiom schemas with semantic side conditions. The axioms of $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU are obtained by replacing each axiom schema by all its instances. Owing to the presence of axiom schemas with semantic side conditions, the axioms of $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU include proper axioms and axioms with semantic side conditions. Therefore, semantic side conditions take part in the translation of the axioms as well. The instances of TSU5, TSU6, and TSU7 are the only axioms of $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU with semantic side conditions. These semantic side conditions, being of the form $H(\langle m \rangle) = r$, are looked upon as elements of C_{at}.

Consider the set that consists of:

- all equations $t_1 = t_2$, where t_1 and t_2 are terms of sort **T**;
- all conditional equations $E \Rightarrow t_1 = t_2$, where t_1 and t_2 are terms of sort **T** and E is a set of equations $t'_1 = t'_2$ where t'_1 and t'_2 are terms of sort **T**;
- all expressions $t_1 = t_2$ if ϕ, where t_1 and t_2 are terms of sort **T** and $\phi \in \mathsf{C}_{\mathsf{at}}$.

We define a translation function $[\![\_]\!]$ from this set to the set of all equations of $\mathrm{ACP}^{\mathrm{c}}$ with action renaming and guarded recursion as follows:

$$\begin{aligned}
[\![t_1 = t_2]\!] &= [\![t_1]\!] = [\![t_2]\!] \,, \\
[\![E \Rightarrow t_1 = t_2]\!] &= \{[\![t_1']\!] = [\![t_2']\!] \mid t_1' = t_2' \in E\} \Rightarrow [\![t_1]\!] = [\![t_2]\!] \,, \\
[\![t_1 = t_2 \mathsf{\ if\ } \phi]\!] &= \mathsf{CE}_{h_{\Phi \cup \{\phi\}}}([\![t_1]\!]) = \mathsf{CE}_{h_{\Phi \cup \{\phi\}}}([\![t_2]\!]) \,,
\end{aligned}$$

where

$$\Phi = \{\textstyle\bigwedge_{r \in \mathcal{R}} \neg\, (H(\langle m \rangle) = r \wedge \bigvee_{r' \in \mathcal{R} \setminus \{r\}} H(\langle m \rangle) = r') \mid H \in \mathcal{S}, m \in \mathcal{M}\} \,.$$

Here h_Ψ is a function on conditions of $\mathrm{ACP}^{\mathrm{c}}$ that preserves $\bot$, $\top$, $-$, $\sqcup$ and $\sqcap$ and satisfies $h_\Psi(\zeta) = \top$ iff ζ corresponds to a proposition derivable from Ψ and $h_\Psi(\zeta) = \bot$ iff $-\zeta$ corresponds to a proposition derivable from Ψ.[6]

Theorem 1 (Preservation). *Let ax be an axiom of* $\mathrm{TA}_{\mathrm{spt}}$+REC+TSU*. Then $[\![ax]\!]$ is derivable from the axioms of* $\mathrm{ACP}^{\mathrm{c}}$ *with action renaming and guarded recursion.*

Proof. The proof is straightforward. In [7], we outline the proof for axiom TSU5. The other axioms are proved in a similar way. □

8 Execution Architectures for Fragmented Programs

An analytic execution architecture in the sense of [11] is a model of a hypothetical execution environment for sequential programs that is designed for the purpose of explaining how a program may be executed. An analytic execution architecture makes explicit the interaction of a program with the components of its execution environment. The notion of analytic execution architecture defined in [11] is suited to sequential programs that have not been split into fragments. In this section, we discuss analytic execution architectures suited to sequential programs that have been split into fragments.

The notion of analytic execution architecture from [11] is defined in the setting of program algebra. In [4], a thread extraction operation $|\_|$ is defined which gives, for each program considered in program algebra, the thread that is taken for the behaviour exhibited by the program on execution. In the case of programs that have been split into fragments, additional instructions for switching over execution to another program fragment are needed. We assume that a collection of program fragments between which execution can be switched takes the form of a sequence, called an program fragment vector, and that there is an additional instruction $\#\#\#i$ for each $i \in \mathbb{N}$. Switching over execution to the i-th program fragment in the program fragment vector is effected by executing the instruction $\#\#\#i$. If i equals 0 or i is greater than the length of the program fragment

[6] Here we use "corresponds to" for the wordy "is isomorphic to the equivalence class with respect to logical equivalence of" (see also [6]).

vector, execution of $\#\#\#i$ results in deadlock. We extend thread extraction as follows:

$$|\#\#\#i| = \mathsf{S}i \;, \qquad |\#\#\#i \,;\, x| = \mathsf{S}i \;.$$

An analytic execution architecture for programs that have been split into fragments consists of a component containing a program fragment, a component containing a program fragment vector and a number of service components. The component containing a program fragment is capable of processing instructions one at a time, issuing appropriate requests to service components and awaiting replies from service components as described in [11] in so far as instructions other than switch-over instructions are concerned. This implies that, for each service component, there is a channel for communication between the program fragment component and that service component and that foci are used as names of those channels. In the case of a switch-over instruction, the component containing a program fragment is capable of loading the program fragment to which execution must be switched from the component containing a program fragment vector.

The analytic execution architecture made up of a component containing the program fragment P, a component containing the program fragment vector $\alpha = \langle P_1 \rangle \frown \ldots \frown \langle P_n \rangle$, and service components $H_1, \ldots, H_k$ with channels named $f_1, \ldots, f_k$, respectively, is described by the thread

$$\boxplus(|P|, \langle |P_1| \rangle \frown \ldots \frown \langle |P_n| \rangle) \,/_{f_1}\, H_1 \ldots /_{f_k}\, H_k \;.$$

In the case where instructions of the form $\#\#\#i$ do not occur in P,

$$[\![\boxplus(|P|, \langle |P_1| \rangle \frown \ldots \frown \langle |P_n| \rangle) \,/_{f_1}\, H_1 \ldots /_{f_k}\, H_k]\!]$$

agrees with the process-algebraic description given in [11] of the analytic execution architecture made up of a component containing the program P and service components $H_1, \ldots, H_k$ with channels named $f_1, \ldots, f_k$, respectively.

9 Conclusions

We have developed a theory of the behaviours exhibited by sequential programs on execution that covers the case where the programs have been split into fragments and have used it to describe analytic execution architectures suited for such programs. It happens that the resulting description is terse. We have also shown that threads and services as considered in this theory can be viewed as processes that are definable over an extension of ACP with conditions. Threads and services are introduced for pragmatic reasons only: describing them as general processes is awkward. For example, the description of analytic execution architectures suited for programs that have been split into fragments would no longer be terse if ACP with conditions had been used.

The object pursued with the line of research that we have carried on with this paper is the development of a theoretical understanding of the concepts

sequential program and sequential program behaviour. We regard the work presented in this paper also as a preparatory step in the development of a theoretical understanding of the concept operating system. In systems resulting from contemporary programming, we find distributed multi-threading and threads that are program behaviour fragments. For that reason, it is an interesting option for future work to combine the theory of distributed strategic interleaving developed in [8] with the theory developed in this paper.

References

1. J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.
2. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings 30th ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
3. J. A. Bergstra, I. Bethke, and A. Ponse. Decision problems for pushdown threads. *Acta Informatica*, 44(2):75–90, 2007.
4. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
5. J. A. Bergstra and C. A. Middelburg. Instruction sequences with indirect jumps. To appear in *Scientific Annals of Computer Science*. Preliminary version: Electronic Report PRG0709, Programming Research Group, University of Amsterdam.
6. J. A. Bergstra and C. A. Middelburg. Splitting bisimulations and retrospective conditions. *Information and Computation*, 204(7):1083–1138, 2006.
7. J. A. Bergstra and C. A. Middelburg. Thread algebra with multi-level strategies. *Fundamenta Informaticae*, 71(2/3):153–182, 2006.
8. J. A. Bergstra and C. A. Middelburg. Distributed strategic interleaving with load balancing. *Future Generation Computer Systems*, 2007. In press, doi: 10.1016/j.future.2007.08.001.
9. J. A. Bergstra and C. A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007.
10. J. A. Bergstra and C. A. Middelburg. Thread extraction for polyadic instruction sequences. Electronic Report PRG0803, Programming Research Group, University of Amsterdam, February 2008.
11. J. A. Bergstra and A. Ponse. Execution architectures for program algebra. *Journal of Applied Logic*, 5:170–192, 2007.
12. W. J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, Berlin, 2000.
13. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.
14. R. Milner. *Communication and Concurrency*. Prentice-Hall, Englewood Cliffs, 1989.
15. A. Ponse and M. B. van der Zwaag. An introduction to program and thread algebra. In A. Beckmann et al., editors, *CiE 2006*, volume 3988 of *Lecture Notes in Computer Science*, pages 445–458. Springer-Verlag, 2006.

16. D. Sannella and A. Tarlecki. Algebraic preliminaries. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 13–30. Springer-Verlag, Berlin, 1999.
17. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, Amsterdam, 1990.

Electronic Reports Series of the Programming Research Group

Within this series the following reports appeared.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Electronic Report Series

Programming Research Group
Faculty of Science
University of Amsterdam

Kruislaan 403
1098 SJ Amsterdam
the Netherlands

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