DOI 10.1007/s00165-011-0178-3

Formal Aspects of Computing (2011) 23: 567–583

Formal Aspects of Computing

Thread algebra for poly-threading

J. A. Bergstra and C. A. Middelburg

Faculty of Science, Informatics Institute, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands.
E-mail: C.A.Middelburg@uva.nl

Abstract. It is a fact of life that sequential programs are often fragmented. Consequently, fragmented program behaviours are frequently found. We consider this phenomenon in the setting of thread algebra. We extend basic thread algebra with poly-threading, the barest mechanism for sequencing of threads that are taken for program fragment behaviours. This mechanism is the counterpart of program overlaying at the level of program behaviours. We relate the resulting theory to the process theory known as ACP and use it to describe analytic execution architectures suited for fragmented programs. We also consider the case where the steps of fragmented program behaviours are interleaved in the ways of non-distributed and distributed multi-threading.

Keywords: Poly-threading, Thread algebra, Process algebra, Execution architecture, Non-distributed multi-threading, Distributed multi-threading

1. Introduction

In [BM08c], we considered fragmentation of sequential programs that take the form of instruction sequences in the setting of program algebra [BL02]. 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. An important reason for fragmentation of programs is that the execution architecture at hand to execute them sets bounds to the size of programs.

In [BL02], 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. [BBP07, BM07a, PvdZ06]). In this line of research, the view is taken that the behaviour exhibited by a sequential program on execution takes the form of a thread as considered in basic thread algebra [BL02].[1] With the current paper, we carry on this line of research. Therefore, we consider program fragment behaviours 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 fragment behaviours. This mechanism is called poly-threading. Poly-threading is the counterpart of program overlaying [Pan68] at the level of threads. Until virtual memory was implemented in mainstream operating systems, program overlaying was commonly used in many applications to deal with the prevailing constraints on the size of programs (see e.g. [Stu78, Car84]). In embedded systems, it is usually important to use memory

Correspondence and offprint requests to: C. A. Middelburg, E-mail: C.A.Middelburg@uva.nl.

[1] In [BL02], basic thread algebra is introduced under the name basic polarized process algebra. Prompted by the development of thread algebra [BM07b], which is a design on top of it, basic polarized process algebra has been renamed to basic thread algebra.

efficiently. However, many embedded systems are real-time systems and the use of virtual memory makes it insufficiently predictable whether a real-time system meets the deadlines set for it. Therefore, program overlaying is still popular in embedded systems (see e.g. [CPC10]). Moreover, with the advent of multi-core processors, there seems to be a renewed interest in program overlaying in the area of operating systems (see e.g. [M+05]). It happens that in computer science since the introduction of program overlaying more than 40 years ago no serious attention has been paid to the explanation of program overlaying at a semantic level. This state of affairs forms our motivation to have a closer look at program overlaying at the level of program behaviours.

Inherent in the behaviour exhibited by a program on execution is that it does certain steps for the purpose of interacting with some service provided by an execution environment. In the setting of thread algebra, the use mechanism is introduced in [BM07b] to allow for this kind of interaction. Poly-threading supports the initialization of one of the services used every time a thread is started up. With 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 thread and service look to be special cases of a more general notion of process. Therefore, it is interesting to know how threads and services as considered in the extension of basic thread algebra with poly-threading relate to processes as considered in theories about concurrent processes such as ACP [BW90], CCS [Mil89] and CSP [Hoa85]. We show that threads and services as considered in the extension of basic thread algebra with poly-threading introduced in this paper can be viewed as processes that are definable over ACP. Because ACP is much better known, this may be helpful in forming a better idea of the extension of basic thread algebra with poly-threading.

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 [BP07] is suited to sequential programs that have not been split into fragments. We use the extension of basic thread algebra with poly-threading to describe analytic execution architectures suited to sequential programs that have been split into fragments.

In systems resulting from contemporary programming, we find non-distributed and distributed multi-threading and threads that are program fragment behaviours, the latter primarily by the advent of multi-core processors (see e.g. [CPC10]). For that reason, it is interesting to combine the theories of non-distributed and distributed strategic interleaving developed in [BM07b] and [BM08a], respectively, with the extension of basic thread algebra with poly-threading. We take up the combination by introducing poly-threading covering variations of the simplest form of interleaving for non-distributed and distributed multi-threading considered in [BM07b, BM08a].

The line of research carried on in this paper has two main themes: the theme of instruction sequences and the theme of threads. Both [BM08c] and the current paper are concerned with program fragmentation, but [BM08c] 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.

This paper is organized as follows. First, we review basic thread algebra and the use mechanism (Sects. 2, 3). Next, we extend basic thread algebra with poly-threading and show how external thread selections in poly-threading can be internalized (Sects. 4 , 5). Following this, we review ACP and relate the extension of basic thread algebra with poly-threading to ACP (Sects. 6, 7). Then, we discuss analytic execution architectures suited for programs that have been fragmented (Sect. 8). After that, we introduce forms of interleaving suited for non-distributed and distributed multi-threading that cover poly-threading (Sects. 9, 10, 11). Finally, we make some concluding remarks (Sect. 12).

Up to and including Sect. 8, this paper is a revision of [BM08b]. In that paper, the term "sequential poly-threading" stands for "poly-threading in a setting where multi-threading or any other form of concurrency is absent". We conclude in hindsight that the use of this term is unfortunate and do not use it in the current paper.

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*.

Table 1. Axiom of BTA

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

Table 2. Axioms for guarded recursion

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

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 Sects. 3, 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 *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. [ST99, Wir90]). 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: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

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 [BB03]. 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 $\langle X \mid E \rangle$. Moreover, we add the axioms for guarded recursion given in Table 2 to BTA, where 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 $\langle Y \mid E \rangle$.[2] 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.

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

[2] Throughout the paper, we use the symbol $\Rightarrow$ for implication. We use the precedence conventions that the equality symbol binds stronger than the implication symbol.

Table 3. Axioms for use

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

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.[3]

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:

$$\forall\, \rho \in \mathcal{M}^+, m \in \mathcal{M} \bullet (H(\rho) = \mathsf{B} \Rightarrow H(\rho \frown \langle m \rangle) = \mathsf{B}).$$[4]

B stands for blocked. It is used to indicate that a request to process a command is rejected.

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 $\frac{\partial}{\partial m} H(\rho) = H(\langle m \rangle \frown \rho)$.

A service H can be understood as follows:

- if $H(\langle m \rangle) \neq \mathsf{B}$, then the request to process m is accepted by the service, the reply is $H(\langle m \rangle)$, and the service proceeds as $\frac{\partial}{\partial m} H$;
- if $H(\langle m \rangle) = \mathsf{B}$, then the request to process m is not accepted by the service.

For each $f \in \mathcal{F}$, we introduce the *use* operator $\_ \mathbin{/_f} \_ : \mathbf{T} \times \mathbf{S} \to \mathbf{T}$. Intuitively, $p \mathbin{/_f} H$ 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 internal action tau that it is left as a trace of the processed action could be abstracted from, but this would exclude analysis to which the presence of internal activity is relevant.

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.

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.

[3] This version of the use mechanism was first introduced in [BM07b]. In later papers, it is also called thread-service composition.

[4] 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.

4. 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 $\mathrm{TA}^{\mathrm{pt}}$.

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 $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.

$\mathrm{TA}^{\mathrm{pt}}$ has the sort $\mathbf{T}$ of BTA and in addition the sort $\mathbf{TV}$ of *thread vectors*. To build terms of sort $\mathbf{T}$, $\mathrm{TA}^{\mathrm{pt}}$ 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 *poly-threading* operator $\boxplus : \mathbf{T} \times \mathbf{TV} \to \mathbf{T}$;
- for each $k \in \mathbb{N}^+$, [5] 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 $\mathbf{TV}$, $\mathrm{TA}^{\mathrm{pt}}$ has the following constants and operators:

- the *empty thread vector* constant $\langle\rangle : \mathbf{TV}$;
- the *singleton thread vector* operator $\langle\_\rangle : \mathbf{T} \to \mathbf{TV}$;
- the *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, β, γ.

In the context of the 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.

[5] We write $\mathbb{N}^+$ for the set $\{n \in \mathbb{N} \mid n > 0\}$. Throughout the paper, we use the convention that k and n stand for arbitrary elements of $\mathbb{N}^+$ and $\mathbb{N}$, respectively.

Table 4. Axioms for poly-threading

$\between(\mathsf{S}, \alpha) = \mathsf{S}$		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
$\between(\mathsf{E}, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle) =$ $\Box_k(\mathsf{tls.init} \circ \between(x_1, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle), \ldots, \mathsf{tls.init} \circ \between(x_k, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle))$		SPT6
$\between(\mathsf{E}, \langle\,\rangle) = \mathsf{D}$		SPT7

Moreover, the basic action 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 Si and E produce in the case where they occur outside the context of the poly-threading operator.

The 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 Si 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 external choice between k threads. An external choice between a number of threads is a choice where the threads concerned have no control of the choice. Let $p_1, \ldots, p_k$ be closed terms of sort $\mathbf{T}$. Then $\Box_k(p_1, \ldots, p_k)$ behaves as the outcome of an external choice between $p_1, \ldots, p_k$ and D.

TA$^{\mathrm{pt}}$ 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 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 $\Box_k$ are expressed by the following disjunction of equations: $\bigvee_{i\in[1,k]} \Box_k(x_1, \ldots, x_k) = x_i \vee \Box_k(x_1, \ldots, x_k) = \mathsf{D}$. [6]

To be fully precise, we should give axioms concerning the constants and operators to build terms of the sort $\mathbf{TV}$ as well. We refrain from doing so because the constants and operators concerned are the usual ones for sequences. Similar remarks apply to the sort $\mathbf{DTV}$ introduced later and will not be repeated.

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

The use mechanism can be added to TA$^{\mathrm{pt}}$ as it is added to BTA in Sect. 3. Let T stand for either TA$^{\mathrm{pt}}$ or TA$^{\mathrm{pt}}$+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 TA$^{\mathrm{pt}}$ 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.

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, …, 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 $\mathcal{M}$.

[6] 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

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

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

$$\boxplus(\rho(p), \langle \rho(p_1) \rangle \frown \ldots \frown \langle \rho(p_k) \rangle \frown \langle \mathsf{ext.sel} \trianglerighteq_k (\mathsf{S1}, \ldots, \mathsf{S}k) \rangle) \,,$$

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 $\mathsf{ext} \in \mathcal{F}$ and $\mathsf{sel} \in \mathcal{M}$.

Postconditional switching is not really necessary for internalization. Let $k_1 = \lfloor k/2 \rfloor$, $k_2 = \lfloor k_1/2 \rfloor$, $k_3 = \lfloor (k - k_1)/2 \rfloor$, ... 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. Algebra of Communicating Processes

In Sect. 7, we will investigate the connections of threads and services with processes as considered in ACP-style process algebras. We will focus on ACP (Algebra of Communicating Processes), which was first presented in [BK84]. In this section, we shortly review ACP. A comprehensive introduction to ACP can be found in [BW90, Fok00].

ACP has one sort: the sort $\mathbf{P}$ of *processes*. In ACP, it is assumed that there are a fixed but arbitrary set A of actions with $\delta \notin \mathsf{A}$ 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 $\mathbf{P}$, $a \in \mathsf{A}$, and $H \subseteq \mathsf{A}$. Intuitively, the constants and operators to build terms of sort $\mathbf{P}$ can be explained as follows:

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

The operators $\lfloor\!\lfloor$ and $|$ are of an auxiliary nature. They are needed to axiomatize ACP. The axioms of ACP are given in e.g. [Fok00].

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 $\mathcal{I} = \emptyset$.

A process is considered definable over ACP if there exists a guarded recursive specification over ACP that has that process as its solution.

A *recursive specification* over ACP is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables of sort $\mathbf{P}$ 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 ACP. 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 ACP in either direction and/or the equations in E except the equation $X = t_X$ from left to right. We only consider models of ACP in which guarded recursive specifications have unique solutions, such as the projective limit model and the finitely branching graph model presented in [BW90].

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 have the same shape as in the case of BTA (see Table 2), but now X, t_X and E stand for an arbitrary variable of sort $\mathbf{P}$, an arbitrary ACP term of sort $\mathbf{P}$ and an arbitrary guarded recursive specification over ACP, respectively.

In order to express the use operators, we need an extension of ACP 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 given in e.g. [Fok00]. We write $\rho_{a' \mapsto a''}$ for the renaming operator ρ_g with g defined by $g(a') = a''$ and $g(a) = a$ if $a \neq a'$.

7. Threads, services and ACP-definable processes

Because it may be helpful in forming a better idea of $\mathrm{TA}^{\mathrm{pt}}$+REC+TSU, we relate in this section threads and services as considered in $\mathrm{TA}^{\mathrm{pt}}$+REC+TSU to processes that are definable over ACP with action renaming.

For that purpose, A and | 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 $\mathsf{a} \in \mathsf{A}$, $f \in \mathcal{F}$, $d \in \mathcal{M} \cup \mathcal{R}$, $m \in \mathcal{M}$, $r \in \mathcal{R}$ and $n \in \mathbb{N}$:

$$\begin{aligned}&\mathrm{s}_f(d) \mid \mathrm{r}_f(d) = \mathrm{i}\ , && && \mathrm{stop} \mid \overline{\mathrm{stop}} = \mathrm{stop}^*\ , && \\ &\mathrm{s}_f(d) \mid a = \delta && \text{if } a \neq \mathrm{r}_f(d)\ , && \mathrm{stop} \mid a = \delta && \text{if } a \neq \overline{\mathrm{stop}}\ , \\ &a \mid \mathrm{r}_f(d) = \delta && \text{if } a \neq \mathrm{s}_f(d)\ , && a \mid \overline{\mathrm{stop}} = \delta && \text{if } a \neq \mathrm{stop}\ ,\end{aligned}$$

$$\begin{aligned}&\mathrm{s}_{\mathrm{ext}}(n) \mid \mathrm{r}_{\mathrm{ext}}(n) = \mathrm{i}\ , && && \mathrm{i} \mid a = \delta\ , \\ &\mathrm{s}_{\mathrm{ext}}(n) \mid a = \delta && \text{if } a \neq \mathrm{r}_{\mathrm{ext}}(n)\ , && \mathrm{s}_{\mathrm{serv}}(r) \mid a = \delta\ , \\ &a \mid \mathrm{r}_{\mathrm{ext}}(n) = \delta && \text{if } a \neq \mathrm{s}_{\mathrm{ext}}(n)\ , && a \mid \mathrm{r}_{\mathrm{serv}}(m) = \delta\ .\end{aligned}$$

The actions of the forms $\mathrm{s}_f(d)$ and $\mathrm{r}_f(d)$ are used for communication between threads and services, the actions of the forms $\mathrm{s}_{\mathrm{ext}}(n)$ and $\mathrm{r}_{\mathrm{ext}}(n)$ are used for communication between threads and their environment, the actions stop and $\overline{\mathrm{stop}}$ are used for synchronization between threads and services on termination, and the actions stop^* and i are the actions that remain as the result of synchronizations on termination and communications, respectively. The actions of the forms $\mathrm{s}_{\mathrm{serv}}(r)$ and $\mathrm{r}_{\mathrm{serv}}(m)$ are actions that are intended to be replaced by actions for communication between threads and services.

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

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

Table 6. Definition of translation functions

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

The sets A_f and the functions R_f play a part in expressing the use operators in terms of the operators of ACP with action renaming. The sets A_f are used to encapsulate actions for communication between threads and services from communication with actions coming from the environment. The functions R_f are used to replace actions of the forms $\mathrm{s}_{\mathrm{serv}}(r)$ and $\mathrm{r}_{\mathrm{serv}}(m)$ by actions for communication between threads and services.

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 $\alpha = \langle p_1 \rangle \frown \ldots \frown \langle p_n \rangle$, and let $i \in [1, n]$. Then we write $\alpha[i]$ for p_i.

We proceed with relating threads and services as considered in TA$^{\mathrm{pt}}$+REC+TSU to processes definable over ACP with action renaming. The underlying idea is that threads and services can be viewed as processes that are definable over ACP with action renaming. We define those processes by means of a translation function $\llbracket \_ \rrbracket$ from the set of all terms of sort $\mathbf{T}$ to the set of all functions from the set of all terms of sort $\mathbf{TV}$ to the set of all terms of sort $\mathbf{P}$ and a translation function $\llbracket \_ \rrbracket$ 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 $\Delta(H)$ is inductively defined as follows:

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

Let p be a term of sort $\mathbf{T}$. Then the *process algebraic interpretation* of p is $\llbracket p \rrbracket(\langle\,\rangle)$. Henceforth, we write $\llbracket \_ \rrbracket$ for the translation function $\llbracket \_ \rrbracket(\langle\,\rangle)$.

The translation function $\llbracket \_ \rrbracket$ preserves the axioms of TA$^{\mathrm{pt}}$ + REC + TSU. Roughly speaking, this means that the translations of these axioms are derivable from the axioms of ACP with action renaming and guarded recursion. Before we make this fully precise, we have a closer look at the axioms of TA$^{\mathrm{pt}}$ + 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 TA$^{\mathrm{pt}}$ + 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 TA$^{\mathrm{pt}}$ + REC + TSU include proper axioms and axioms with semantic side conditions. The instances of TSU5, TSU6, and TSU7 are the only axioms of TA$^{\mathrm{pt}}$ + REC + TSU with semantic side conditions. These semantic side conditions concerned are of the form $H(\langle m \rangle) = r$.

Consider the set that consists of:

- all equations $t_1 = t_2$, where t_1 and t_2 are terms of sort $\mathbf{T}$;
- all conditional equations $E \Rightarrow t_1 = t_2$, where t_1 and t_2 are terms of sort $\mathbf{T}$ and E is a set of equations $t_1' = t_2'$ where t_1' and t_2' are terms of sort $\mathbf{T}$.

We define a translation function $\llbracket \_ \rrbracket$ from this set to the set of all equations of ACP with action renaming and guarded recursion as follows:

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

Theorem (Preservation)

1. *Let ϕ be a proper axiom of* $\mathrm{TA}^{\mathrm{pt}}$+REC+TSU. *Then $[\![\phi]\!]$ is derivable from the axioms of* ACP *with action renaming and guarded recursion.*
2. *Let ϕ* if *ψ be an axiom of* $\mathrm{TA}^{\mathrm{pt}}$+REC+TSU *with semantic side condition. Then $[\![\phi]\!]$ is derivable from the axioms of* ACP *with action renaming and guarded recursion under the assumption that ψ holds.*

Proof. The proof is straightforward. We sketch the proof for axiom TSU5, writing E_H for the guarded recursive specification $\{X_{H'} = \sum_{m \in \mathcal{M}} \mathrm{r}_{\mathrm{serv}}(m) \cdot \mathrm{s}_{\mathrm{serv}}(H'(\langle m \rangle)) \cdot X_{\frac{\partial}{\partial m} H'} + \overline{\mathsf{stop}} \mid H' \in \Delta(H)\}$. By the definition of the translation function $[\![\_]\!]$, it must be shown that $[\![(x \trianglelefteq f.m \trianglerighteq y) /_f H]\!] = [\![\mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)]\!]$ is derivable under the assumption that $H(\langle m \rangle) = \mathsf{T}$ holds. In outline, this goes as follows:

$$
\begin{aligned}
&[\![(x \trianglelefteq f.m \trianglerighteq y) /_f H]\!] \\
&\quad = \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}, \overline{\mathrm{stop}}\}}(\partial_{A_f}(\mathrm{s}_f(m) \cdot (\mathrm{r}_f(\mathsf{T}) \cdot x + \mathrm{r}_f(\mathsf{F}) \cdot y) \parallel \rho_{R_f}(\langle X_H | E_H \rangle))))) \\
&\quad = \mathrm{i} \cdot \mathrm{i} \cdot \rho_{\mathrm{stop}^* \mapsto \mathrm{stop}}(\partial_{\{\mathrm{stop}, \overline{\mathrm{stop}}\}}(\partial_{A_f}(x \parallel \rho_{R_f}(\langle X_{\frac{\partial}{\partial m} H} | E_H \rangle)))) \\
&\quad = [\![\mathsf{tau} \circ (x /_f \frac{\partial}{\partial m} H)]\!] \,.
\end{aligned}
$$

In the first and third step, we apply defining equations of $[\![\_]\!]$. In the second step, we apply axioms of ACP with action renaming and guarded recursion, and use the assumption that $H(\langle m \rangle) = \mathsf{T}$. □

The gist of the preservation theorem is that the translation function $[\![\_]\!]$ represents a theory interpretation. In logic, however, the concept of a theory interpretation does not take semantic side conditions into account.

8. Execution architectures for fragmented programs

An analytic execution architecture in the sense of [BP07] 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 [BP07] 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 in the current setting. Thus, we illustrate an application of the thread algebra for poly-threading developed so far.

The notion of analytic execution architecture from [BP07] is defined in the setting of program algebra. In [BL02], 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 vector, execution of $\#\#\#i$ results in deadlock. The instruction $\#\#\#i$ is reminiscent of the system call `exec()` of Unix-like operating systems. 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 [BP07] 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.

Table 7. Axioms for poly-threaded cyclic interleaving

$\parallel_{\boxplus}(\langle\,\rangle, \alpha) = \mathsf{S}$		PCI1
$\parallel_{\boxplus}(\langle \mathsf{S}\rangle \frown \beta, \alpha) = \parallel_{\boxplus}(\beta, \alpha)$		PCI2
$\parallel_{\boxplus}(\langle \mathsf{D}\rangle \frown \beta, \alpha) = \mathsf{S}_{\mathsf{D}}(\parallel_{\boxplus}(\beta, \alpha))$		PCI3
$\parallel_{\boxplus}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \beta, \alpha) = \parallel_{\boxplus}(\beta \frown \langle x\rangle, \alpha) \trianglelefteq a \trianglerighteq \parallel_{\boxplus}(\beta \frown \langle y\rangle, \alpha)$		PCI4
$\parallel_{\boxplus}(\langle \mathsf{S}i\rangle \frown \beta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) =$		
$\quad \mathsf{tls.init} \circ \parallel_{\boxplus}(\beta \frown \langle x_i\rangle, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle)$	if $1 \leq i \leq n$	PCI5
$\parallel_{\boxplus}(\langle \mathsf{S}i\rangle \frown \beta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) = \mathsf{S}_{\mathsf{D}}(\parallel_{\boxplus}(\beta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle))$	if $i = 0 \vee i > n$	PCI6
$\parallel_{\boxplus}(\langle \mathsf{E}\rangle \frown \beta, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle) =$		
$\quad \Box_k(\mathsf{tls.init} \circ \parallel_{\boxplus}(\beta \frown \langle x_1\rangle, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle), \ldots, \mathsf{tls.init} \circ \parallel_{\boxplus}(\beta \frown \langle x_k\rangle, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle))$		PCI7
$\parallel_{\boxplus}(\langle \mathsf{E}\rangle \frown \beta, \langle\,\rangle) = \mathsf{S}_{\mathsf{D}}(\parallel_{\boxplus}(\beta, \langle\,\rangle))$		PCI8

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 [BP07] 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. Poly-threaded strategic interleaving

In this section, we take up the extension of $\mathrm{TA}^{\mathrm{pt}}$ with a form of interleaving suited for multi-threading. Multi-threading refers to the concurrent existence of several threads in a program under execution. Multi-threading is provided by contemporary programming languages such as Java [GJSB00] and C# [HWG03]. Arbitrary interleaving, on which ACP [BW90], CCS [Mil89] and CSP [Hoa85] are based, is not an appropriate abstraction when dealing with multi-threading. In the case of multi-threading, some deterministic interleaving strategy is used. In [BM07b], we introduced a number of plausible deterministic interleaving strategies for multi-threading. We proposed to use the phrase "strategic interleaving" for the more constrained form of interleaving obtained by using such a strategy. In this section, we consider a poly-threading covering variation of the simplest interleaving strategy introduced in [BM07b], namely pure *cyclic interleaving*.

As in [BM07b], it is assumed that the collection of threads to be interleaved takes the form of a thread vector. In the poly-threaded case, cyclic interleaving basically operates as follows: at each stage of the interleaving, the first thread in the thread vector gets a turn to perform a basic action or to switch over to another thread and then the thread vector undergoes cyclic permutation. We mean by cyclic permutation of a thread vector that the first thread in the thread vector becomes the last one and all others move one position to the left. If one thread in the thread vector deadlocks, the whole does not deadlock till all others have terminated or deadlocked. An important property of cyclic interleaving is that it is fair, i.e. there will always come a next turn for all active threads. Other plausible interleaving strategies are treated in [BM07b]. They can also be adapted to the poly-threaded case.

The extension of $\mathrm{TA}^{\mathrm{pt}}$ with cyclic interleaving is called $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{si}}$. It has the sorts $\mathbf{T}$ and $\mathbf{TV}$ of $\mathrm{TA}^{\mathrm{pt}}$. To build terms of sort $\mathbf{T}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{si}}$ has the constants and operators of $\mathrm{TA}^{\mathrm{pt}}$ to build terms of sort $\mathbf{T}$ and in addition the following operator:

- the *poly-threaded cyclic strategic interleaving* operator $\parallel_{\boxplus} : \mathbf{TV} \times \mathbf{TV} \to \mathbf{T}$.

To build terms of sort $\mathbf{TV}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{si}}$ has the constants and operators of $\mathrm{TA}^{\mathrm{pt}}$ to build terms of sort $\mathbf{TV}$.

$\mathrm{TA}^{\mathrm{pt}}_{\mathrm{si}}$ has the axioms of $\mathrm{TA}^{\mathrm{pt}}$ and in addition the axioms given in Tables 7 and 8. In these tables, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. The axioms from Table 7 express that threads are interleaved as described above. In these axioms, the auxiliary *deadlock at termination* operator S_{D} occurs. The axioms from Table 8 show that this operator serves to turn termination into deadlock.

Table 8. Axioms for deadlock at termination

$\mathsf{S_D}(\mathsf{S}) = \mathsf{D}$	S2D1
$\mathsf{S_D}(\mathsf{D}) = \mathsf{D}$	S2D2
$\mathsf{S_D}(x \trianglelefteq a \trianglerighteq y) = \mathsf{S_D}(x) \trianglelefteq a \trianglerighteq \mathsf{S_D}(y)$	S2D3
$\mathsf{S_D}(\mathsf{S}i) = \mathsf{S}i$	S2D4
$\mathsf{S_D}(\mathsf{E}) = \mathsf{E}$	S2D5
$\mathsf{S_D}(\Box_k(x_1, \ldots, x_k)) = \Box_k(\mathsf{S_D}(x_1), \ldots, \mathsf{S_D}(x_k))$	S2D6

Guarded recursion and the use mechanism can be added to $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{si}}$ as they are added to BTA in Sects. 2, 3, respectively.

The axioms for poly-threaded cyclic interleaving given in Table 7 constitute a definition by recursion on the sum of the depths of all threads in the first thread vector with case distinction on the structure of the first thread in that thread vector and, in the case of E, with case distinction on the length of the second thread vector. Hence, it is obvious that the axioms are consistent and that every closed term of the sort $\mathbf{T}$ is derivably equal to one that does not contain the poly-threaded cyclic strategic interleaving operator. Similar remarks apply to the poly-threaded cyclic distributed strategic interleaving operators introduced in subsequent sections and will not be repeated.

10. Poly-threaded distributed strategic interleaving

In this section, we take up the extension of $\mathrm{TA}^{\mathrm{pt}}$ with a form of interleaving suited for distributed multi-threading. This extension concerns a plausible poly-threading covering variation of the simplest form of interleaving for distributed multi-threading considered in [BM08a]. The variation in question may be considered a very simple approximation of the form of interleaving that is found in systems built on multi-core processors and a fair form of scheduling (presumably realized by means of a global scheduler and a local scheduler per core). The simplification is considerable indeed, but we believe that this is inevitable to obtain a manageable theory.

In order to deal with threads that are distributed over the nodes of a network, it is assumed that there is a fixed but arbitrary finite set $\mathcal{L}$ of *locations* such that $\mathcal{L} \subseteq \mathbb{N}$. The set $\mathcal{LA}$ of *located basic actions* is defined by $\mathcal{LA} = \{l.a \mid l \in \mathcal{L} \wedge a \in \mathcal{A}\}$. Henceforth, basic actions will also be called *unlocated basic actions*. The members of $\mathcal{LA} \cup \{l.\mathsf{tau} \mid l \in \mathcal{L}\}$ are referred to as *located actions*.

Performing an unlocated action a is taken as performing a at a location still to be fixed by the distributed interleaving strategy. Performing a located action $l.a$ is taken as performing a at location l.

Threads that perform unlocated actions only are called *unlocated* threads and threads that perform located actions only are called *located* threads. It is assumed that the collection of all threads that exist concurrently at the same location takes the form of a sequence of unlocated threads, called the *local thread vector* at the location concerned. It is also assumed that the collection of local thread vectors that exist concurrently at the different locations takes the form of a sequence of pairs, one for each location, consisting of a location and the local thread vector at that location. Such a sequence is called a *distributed thread vector*.

In the distributed case, cyclic interleaving basically operates the same as in the non-distributed case. In the distributed case, we mean by cyclic permutation of a distributed thread vector that the first thread in the first local thread vector becomes the last thread in the first local thread vector, all other threads in the first local thread vector move one position to the left, the resulting local thread vector becomes the last local thread vector in the distributed thread vector, and all other local thread vectors in the distributed thread vector move one position to the left.

When discussing interleaving strategies on distributed thread vectors, we use the term "current thread" to refer to the first thread in the first local thread vector in a distributed thread vector and we use the term "current location" to refer to the location at which the first local thread vector in a distributed thread vector is.

The extension of $\mathrm{TA}^{\mathrm{pt}}$ with cyclic distributed interleaving is called $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$. It has the sorts $\mathbf{T}$ and $\mathbf{TV}$ of $\mathrm{TA}^{\mathrm{pt}}$ and in addition the following sorts:

- the sort $\mathbf{LT}$ of *located threads*;
- the sort $\mathbf{DTV}$ of *distributed thread vectors*.

Table 9. Definition of the functions app_l

$app_l(x, \langle\rangle) = \langle\rangle$	
$app_l(x, [\gamma]_{l'} \frown \delta) = [\gamma \frown \langle x\rangle]_l \frown \delta$	if $l = l'$
$app_l(x, [\gamma]_{l'} \frown \delta) = [\gamma]_{l'} \frown app_l(x, \delta)$	if $l \neq l'$

Table 10. Axioms for postconditional composition

$u \trianglelefteq l.\mathsf{tau} \trianglerighteq v = u \trianglelefteq l.\mathsf{tau} \trianglerighteq u$	LT1

To build terms of sort $\mathbf{T}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$ has the constants and operators of BTA and in addition the following operators:

- for each $n \in \mathbb{N}$, the *migration postconditional composition* operator $\_ \trianglelefteq \mathsf{mg}(n) \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \to \mathbf{T}$.

To build terms of sort $\mathbf{TV}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$ has the constants and operators of $\mathrm{TA}^{\mathrm{pt}}$ to build terms of sort $\mathbf{TV}$. To build terms of sort $\mathbf{LT}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$ has the following constants and operators:

- the *deadlock* constant $\mathsf{D} : \mathbf{LT}$;
- the *termination* constant $\mathsf{S} : \mathbf{LT}$;
- for each $l \in \mathcal{L}$ and $a \in \mathcal{A}_{\mathsf{tau}}$, the *postconditional composition* operator $\_ \trianglelefteq l.a \trianglerighteq \_ : \mathbf{LT} \times \mathbf{LT} \to \mathbf{LT}$;
- the *deadlock at termination* operator $\mathsf{S}_\mathsf{D} : \mathbf{LT} \to \mathbf{LT}$;
- the *poly-threaded cyclic distributed strategic interleaving* operator $\|_{\boxplus} : \mathbf{DTV} \times \mathbf{TV} \to \mathbf{LT}$.

To build terms of sort $\mathbf{DTV}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$ has the following constants and operators:

- the *empty distributed thread vector* constant $\langle\rangle : \mathbf{DTV}$;
- for each $l \in \mathcal{L}$, the *singleton distributed thread vector* operator $[\_]_l : \mathbf{TV} \to \mathbf{DTV}$;
- the *distributed thread vector concatenation* operator $\frown : \mathbf{DTV} \times \mathbf{DTV} \to \mathbf{DTV}$.

Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{LT}$, including u, v, w, and infinitely many variables of sort $\mathbf{DTV}$, including δ.

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

The overloading of D, S, $\langle\rangle$ and $\frown$ could be resolved, but we refrain from doing so because it is always clear from the context which constant or operator is meant.

Essentially, the sort $\mathbf{DTV}$ includes all sequences of pairs consistingof a location and a local thread vector.[7] The ones that contain a unique pair for each location are the proper distributed thread vectors in the sense that the cyclic distributed interleaving strategy outlined above is intended for them. Improper distributed thread vectors that do not contain duplicate pairs for some location are needed in the axiomatization of this strategy. Improper distributed thread vectors that do contain duplicate pairs for some location appear to have more than one local thread vector at the location concerned. Their exclusion would make it necessary for concatenation of distributed thread vectors to be turned into a partial operator. The cyclic distributed interleaving strategy never turns a proper distributed thread vector into an improper one or the other way round.

The poly-threaded cyclic distributed strategic interleaving operator serves for interleaving of the threads in a proper distributed thread vector according to the strategy outlined above, but with support of explicit thread migration. In the case where a local thread vector of the form $\langle p \trianglelefteq \mathsf{mg}(n) \trianglerighteq q\rangle \frown \gamma$ with $n \in \mathcal{L}$ is encountered as the first local thread vector, γ becomes the last local thread vector in the distributed thread vector and p is appended to the local thread vector at location n. If $n \notin \mathcal{L}$, then $\gamma \frown \langle q\rangle$ becomes the last local thread vector in the distributed thread vector.

In the axioms for cyclic distributed interleaving discussed below, binary functions app_l (where $l \in \mathcal{L}$) from unlocated threads and distributed thread vectors to distributed thread vectors are used. For each $l \in \mathcal{L}$, app_l maps each unlocated thread x and distributed thread vector δ to the distributed thread vector obtained by appending x to the local thread vector at location l in δ. The functions app_l are defined in Table 9.

$\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$ has the axioms of $\mathrm{TA}^{\mathrm{pt}}$ and in addition the axioms given in Tables 10, 11 and 12. In these tables, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. The axioms from Table 11 express that threads are interleaved as described above. The axioms from Tables 10 and 12 are the axioms from Tables 1 and 8 adapted to located threads.

[7] The singleton distributed thread vector operators involve an implicit pairing of their operand with a location.

Table 11. Axioms for poly-threaded cyclic distributed interleaving

$\lVert_{\boxplus}(\langle\,\rangle, \alpha) = \mathsf{S}$		PCDI1
$\lVert_{\boxplus}([\langle\,\rangle]_{l_1} \frown \ldots \frown [\langle\,\rangle]_{l_k}, \alpha) = \mathsf{S}$		PCDI2
$\lVert_{\boxplus}([\langle\,\rangle]_l \frown \delta, \alpha) = \lVert_{\boxplus}(\delta \frown [\langle\,\rangle]_l, \alpha)$		PCDI3
$\lVert_{\boxplus}([\langle \mathsf{S}\rangle \frown \gamma]_l \frown \delta, \alpha) = \lVert_{\boxplus}(\delta \frown [\gamma]_l, \alpha)$		PCDI4
$\lVert_{\boxplus}([\langle \mathsf{D}\rangle \frown \gamma]_l \frown \delta, \alpha) = \mathsf{S}_{\mathsf{D}}(\lVert_{\boxplus}(\delta \frown [\gamma]_l, \alpha))$		PCDI5
$\lVert_{\boxplus}([\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma]_l \frown \delta, \alpha) =$ $\lVert_{\boxplus}(\delta \frown [\gamma \frown \langle x\rangle]_l, \alpha) \trianglelefteq l.a \trianglerighteq \lVert_{\boxplus}(\delta \frown [\gamma \frown \langle y\rangle]_l, \alpha)$		PCDI6
$\lVert_{\boxplus}([\langle \mathsf{S}i\rangle \frown \gamma]_l \frown \delta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) =$ $l.\mathsf{tls.init} \circ \lVert_{\boxplus}(\delta \frown [\gamma \frown \langle x_i\rangle]_l, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle)$	if $1 \leq i \leq n$	PCDI7
$\lVert_{\boxplus}([\langle \mathsf{S}i\rangle \frown \gamma]_l \frown \delta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) =$ $\mathsf{S}_{\mathsf{D}}(\lVert_{\boxplus}(\delta \frown [\gamma]_l, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle))$	if $i = 0 \vee i > n$	PCDI8
$\lVert_{\boxplus}([\langle \mathsf{E}\rangle \frown \gamma]_l \frown \delta, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle) =$ $\Box_k(l.\mathsf{tls.init} \circ \lVert_{\boxplus}(\delta \frown [\gamma \frown \langle x_1\rangle]_l, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle), \ldots,$ $l.\mathsf{tls.init} \circ \lVert_{\boxplus}(\delta \frown [\gamma \frown \langle x_k\rangle]_l, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle))$		PCDI9
$\lVert_{\boxplus}([\langle \mathsf{E}\rangle \frown \gamma]_l \frown \delta, \langle\,\rangle) = \mathsf{S}_{\mathsf{D}}(\lVert_{\boxplus}(\delta \frown [\gamma]_l, \langle\,\rangle))$		PCDI10
$\lVert_{\boxplus}([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle \frown \gamma]_l \frown \delta, \alpha) = l.\mathsf{tau} \circ \lVert_{\boxplus}(app_n(x, \delta \frown [\gamma]_l), \alpha)$	if $n \in \mathcal{L}$	PCDI11
$\lVert_{\boxplus}([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle \frown \gamma]_l \frown \delta, \alpha) = l.\mathsf{tau} \circ \lVert_{\boxplus}(\delta \frown [\gamma \frown \langle y\rangle]_l, \alpha)$	if $n \notin \mathcal{L}$	PCDI12

Table 12. Axioms for deadlock at termination

$\mathsf{S}_{\mathsf{D}}(\mathsf{S}) = \mathsf{D}$	LS2D1
$\mathsf{S}_{\mathsf{D}}(\mathsf{D}) = \mathsf{D}$	LS2D2
$\mathsf{S}_{\mathsf{D}}(u \trianglelefteq a \trianglerighteq v) = \mathsf{S}_{\mathsf{D}}(u) \trianglelefteq a \trianglerighteq \mathsf{S}_{\mathsf{D}}(v)$	LS2D3
$\mathsf{S}_{\mathsf{D}}(\mathsf{S}i) = \mathsf{S}i$	LS2D4
$\mathsf{S}_{\mathsf{D}}(\mathsf{E}) = \mathsf{E}$	LS2D5
$\mathsf{S}_{\mathsf{D}}(\Box_k(u_1, \ldots, u_k)) = \Box_k(\mathsf{S}_{\mathsf{D}}(u_1), \ldots, \mathsf{S}_{\mathsf{D}}(u_k))$	LS2D6

Guarded recursion and the use mechanism can be added to $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$ as they are added to BTA in Sects. 2, 3, respectively.

11. Fragment searching by implicit migration

In Sect. 10, it was assumed that the same program fragment behaviours are available at each location. In the case where this assumption does not hold, distributed interleaving strategies with implicit migration of threads to achieve availability of fragments needed by the threads are plausible. We say that such distributed interleaving strategies take care of fragment searching. In this section, we introduce a variation of the distributed interleaving strategy from Sect. 10 with fragment searching. This results in a theory called $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi,fs}}$.

It is assumed that there is a fixed but arbitrary set $\mathcal{I}$ of *fragment indices* such that $\mathcal{I} = [1, n]$ for some $n \in \mathbb{N}$.

In the case of the distributed interleaving strategy with fragment searching, immediately after the current thread has performed an action, implicit migration of that thread to another location may take place. Whether migration really takes place, depends on the fragments present at the current location. The current thread is implicitly migrated if the following condition is fulfilled: on its next turn, the current thread ought to switch over to a fragment that is not present at the current location. If this conditions is fulfilled, then the current thread will be migrated to the first among the locations where the fragment concerned is present.

To deal with that, we have to enrich distributed thread vectors. The new distributed thread vectors are sequences of triples, one for each location, consisting of a location, the local thread vector at that location, and the set of all indices of fragments that are present at that location.

Table 13. Definition of the functions app'_l

$app'_l(x, \langle\,\rangle) = \langle\,\rangle$	
$app'_l(x, [\gamma]^I_{l'} \frown \delta) = [\gamma \frown \langle x\rangle]^I_l \frown \delta$	if $l = l'$
$app'_l(x, [\gamma]^I_{l'} \frown \delta) = [\gamma]^I_{l'} \frown app'_l(x, \delta)$	if $l \neq l'$

Table 14. Definition of the functions iml', iml and pv

$iml'(i, \langle\,\rangle, l') = l'$	
$iml'(i, [\gamma]^I_l \frown \delta, l') = l$	if $i \in I$
$iml'(i, [\gamma]^I_l \frown \delta, l') = iml'(i, \delta, l')$	if $i \notin I$
$iml([\langle\,\rangle]^I_l \frown \delta) = l$	
$iml([\langle \mathsf{S}\rangle \frown \gamma]^I_l \frown \delta) = l$	
$iml([\langle \mathsf{D}\rangle \frown \gamma]^I_l \frown \delta) = l$	
$iml([\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma]^I_l \frown \delta) = l$	
$iml([\langle \mathsf{S}i\rangle \frown \gamma]^I_l \frown \delta) = l$	if $i \in I$
$iml([\langle \mathsf{S}i\rangle \frown \gamma]^I_l \frown \delta) = iml'(i, \delta, l)$	if $i \notin I$
$iml([\langle \mathsf{E}\rangle \frown \gamma]^I_l \frown \delta) = l$	
$iml([\langle x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y\rangle \frown \gamma]^I_l \frown \delta) = l$	
$pv(\langle\,\rangle) = \langle\,\rangle$	
$pv([\langle\,\rangle]^I_l \frown \delta) = [\langle\,\rangle]^I_l \frown \delta$	
$iml([\langle x\rangle \frown \gamma]^I_l \frown \delta) = l' \Rightarrow pv([\langle x\rangle \frown \gamma]^I_l \frown \delta) = app'_{l'}(x, \delta \frown [\gamma]^I_l)$	

$\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi,fs}}$ has the same sorts as $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$. To build terms of the sorts $\mathbf{T}$, $\mathbf{TV}$ and $\mathbf{LT}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi,fs}}$ has the same constants and operators as $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi}}$. To build terms of sort $\mathbf{DTV}$, $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi,fs}}$ has the following constants and operators:

- the *empty distributed thread vector* constant $\langle\,\rangle : \mathbf{DTV}$;
- for each $l \in \mathcal{L}$ and $I \subseteq \mathcal{I}$, the *singleton distributed thread vector* operator $[\_]^I_l : \mathbf{TV} \to \mathbf{DTV}$;
- the *distributed thread vector concatenation* operator $\frown : \mathbf{DTV} \times \mathbf{DTV} \to \mathbf{DTV}$.

That is, the operator $[\_]_l$ is replaced by the operators $[\_]^I_l$.

Essentially, the sort $\mathbf{TV}$ includes all sequences of unlocated threads. These sequences may serve as local thread vectors and as fragment vectors. The sequences that contain a thread for each fragment index are proper fragment vectors. In the case of fragment vectors that contain more threads, there appear to be inaccessible fragments and in the case of fragment vectors that contain less threads, there appear to be disabled fragments. Inaccessible fragments have no influence on the effectiveness of cyclic distributed interleaving with fragment searching. However, disabled fragments may lead to implicit migration to a location where a switch-over on the next turn is not possible as well. Should this case arise, the next turn will yield deadlock.

In the axioms for cyclic distributed interleaving with fragment searching discussed below, binary functions app'_l (where $l \in \mathcal{L}$) from unlocated threads and distributed thread vectors to distributed thread vectors are used which are similar to the functions app_l used in the axioms for cyclic distributed interleaving without fragment searching given in Sect. 10. The functions app'_l are defined in Table 13.

Moreover, a unary function pv on distributed thread vectors is used which permutes distributed thread vectors cyclicly with implicit migration as outlined above. The function pv is defined using two auxiliary functions:

- a function iml' mapping each fragment index i, distributed thread vector δ and location l to the first location in δ at which the fragment with index i is present if the fragment concerned is present anywhere, and location l otherwise;
- a function iml mapping each non-empty distributed thread vector δ to the first location in δ at which the fragment is present to which the current thread ought to switch over on its next turn if the current thread is in that circumstance and the fragment concerned is present somewhere, and the current location otherwise.

The function pv, as well as the auxiliary functions iml' and iml, are defined in Table 14.

$\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi,fs}}$ has the axioms of $\mathrm{TA}^{\mathrm{pt}}$ and in addition the axioms given in Tables 10, 15 and 12.

Guarded recursion and the use mechanism can be added to $\mathrm{TA}^{\mathrm{pt}}_{\mathrm{dsi,fs}}$ as they are added to BTA in Sects. 2, 3, respectively.

Table 15. Axioms for cyclic distributed interleaving with fragment searching

$\lVert_{\boxplus}(\langle\,\rangle, \alpha) = \mathsf{S}$		PCDIfs1
$\lVert_{\boxplus}([\langle\,\rangle]^I_{i_1} \frown \ldots \frown [\langle\,\rangle]^I_{i_k}, \alpha) = \mathsf{S}$		PCDIfs2
$\lVert_{\boxplus}([\langle\,\rangle]^I_l \frown \delta, \alpha) = \lVert_{\boxplus}(\delta \frown [\langle\,\rangle]^I_l, \alpha)$		PCDIfs3
$\lVert_{\boxplus}([\langle \mathsf{S}\rangle \frown \gamma]^I_l \frown \delta, \alpha) = \lVert_{\boxplus}(\delta \frown [\gamma]^I_l, \alpha)$		PCDIfs4
$\lVert_{\boxplus}([\langle \mathsf{D}\rangle \frown \gamma]^I_l \frown \delta, \alpha) = \mathsf{S_D}(\lVert_{\boxplus}(\delta \frown [\gamma]^I_l, \alpha))$		PCDIfs5
$\lVert_{\boxplus}([\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \gamma]^I_l \frown \delta, \alpha) =$		
$\quad \lVert_{\boxplus}(pv([\langle x\rangle \frown \gamma]^I_l \frown \delta), \alpha) \trianglelefteq l.a \trianglerighteq \lVert_{\boxplus}(pv([\langle y\rangle \frown \gamma]^I_l \frown \delta), \alpha)$		PCDIfs6
$\lVert_{\boxplus}([\langle \mathsf{S}i\rangle \frown \gamma]^I_l \frown \delta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) =$		
$\quad l.\mathsf{tls.init} \circ \lVert_{\boxplus}(pv([\langle x_i\rangle \frown \gamma]^I_l \frown \delta), \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle)$	if $i \in I \cap [1, n]$	PCDIfs7
$\lVert_{\boxplus}([\langle \mathsf{S}i\rangle \frown \gamma]^I_l \frown \delta, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle) =$		
$\quad \mathsf{S_D}(\lVert_{\boxplus}(\delta \frown [\gamma]^I_l, \langle x_1\rangle \frown \ldots \frown \langle x_n\rangle))$	if $i \notin I \cap [1, n]$	PCDIfs8
$\lVert_{\boxplus}([\langle \mathsf{E}\rangle \frown \gamma]^I_l \frown \delta, \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle) =$		
$\quad \Box_k(l.\mathsf{tls.init} \circ \lVert_{\boxplus}(pv([\langle x_1\rangle \frown \gamma]^I_l \frown \delta), \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle), \ldots,$		
$\qquad l.\mathsf{tls.init} \circ \lVert_{\boxplus}(pv([\langle x_k\rangle \frown \gamma]^I_l \frown \delta), \langle x_1\rangle \frown \ldots \frown \langle x_k\rangle))$		PCDIfs9
$\lVert_{\boxplus}([\langle \mathsf{E}\rangle \frown \gamma]^I_l \frown \delta, \langle\,\rangle) = \mathsf{S_D}(\lVert_{\boxplus}(\delta \frown [\gamma]^I_l, \langle\,\rangle))$		PCDIfs10
$\lVert_{\boxplus}([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle \frown \gamma]^I_l \frown \delta, \alpha) = l.\mathsf{tau} \circ \lVert_{\boxplus}(app'_n(x, \delta \frown [\gamma]^I_l), \alpha)$	if $n \in \mathcal{L}$	PCDIfs11
$\lVert_{\boxplus}([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle \frown \gamma]^I_l \frown \delta, \alpha) = l.\mathsf{tau} \circ \lVert_{\boxplus}(pv([\langle y\rangle \frown \gamma]^I_l \frown \delta), \alpha)$	if $n \notin \mathcal{L}$	PCDIfs12

12. 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 ACP. 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 had been used.

We have also taken up the extension of the theory developed to the case where the steps of fragmented program behaviours are interleaved in the ways of non-distributed and distributed multi-threading. This work can be further elaborated on the lines of [BM08a] to cover issues such as prevention from migration for threads that keep locks on shared services, load balancing by means of implicit migration, and the use of implicit migration to achieve availability of services needed by threads.

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 concept of a sequential program and the concept of a sequential program behaviour. We regard the work presented in this paper primarily as a step in the development of a theoretical understanding of the concept of a sequential program behaviour in the presence of program overlaying. We think that such theoretical understanding is important to the development of successful operating systems for real-time embedded systems.

Acknowledgements

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). We thank an anonymous referee for his/her valuable suggestions concerning the presentation of the paper.

References

[BB03] Bergstra JA, Bethke I (2003) Polarized process algebra and program equivalence. In: Baeten JCM, Lenstra JK, Parrow J, Woeginger GJ (eds), Proceedings 30th ICALP, volume 2719 of Lecture Notes in Computer Science. Springer, Berlin pp. 1–21

[BBP07] Bergstra JA, Bethke I, Ponse A (2007) Decision problems for pushdown threads. Acta Informatica 44(2):75–90
[BK84] Bergstra JA, Klop JW (1984) Process algebra for synchronous communication. Inf Control 60(1–3) :109–137
[BL02] Bergstra JA, Loots ME (2002) Program algebra for sequential code. J Logic Algebraic Program 51(2):125–156
[BM07a] Bergstra JA, Middelburg CA (2007) Instruction sequences with indirect jumps. Sci Annal Comput Sci 17 :19–46
[BM07b] Bergstra JA, Middelburg CA (2007) Thread algebra for strategic interleaving. Formal Aspects Comput 19(4):445–474
[BM08a] Bergstra JA, Middelburg CA (2008) Distributed strategic interleaving with load balancing. Future Gener Comput Syst 24(6):530–548
[BM08b] Bergstra JA, Middelburg CA. Thread algebra for sequential poly-threading. Electronic Report PRG0804, Programming Research Group, University of Amsterdam, March 2008. Available at http://www.science.uva.nl/research/prog/publications.html. Also available at http://arxiv.org/: `arXiv:0803.0378v1 [cs.LO]`
[BM08c] Bergstra JA, Middelburg CA Thread extraction for polyadic instruction sequences. Electronic Report PRG0803, Programming Research Group, University of Amsterdam. Available at http://www.science.uva.nl/research/prog/publications.html. Also available at http://arxiv.org/: `arXiv:0802.1578v2 [cs.PL]`
[BP07] Bergstra JA and Ponse A (2007) Execution architectures for program algebra. J Appl Logic 5(1):170–192
[BW90] Baeten JCM, Weijland WP (1990) Process Algebra, volume 18 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge
[Car84] Carlow GD (1984) Architecture of the space shuttle primary avionics software system. Commun ACM 27(9):926–936
[CPC10] Che W, Panda A, Chatha KS (2010) Compilation of stream programs for multicore processors that incorporate scratchpad memories. In DATE '10, pp 1118–1123. IEEE Computer Society Press, Alamitos
[Fok00] Fokkink WJ (2000) Introduction to Process Algebra. Texts in Theoretical Computer Science, An EATCS Series. Springer, Berlin
[GJSB00] Gosling J, Joy B, Steele G, and Bracha G (2000) The Java Language Specification. Addison-Wesley, Reading, MA, second edn
[Hoa85] Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs
[HWG03] Hejlsberg A, Wiltamuth S, and Golde P (2003) C# Language Specification. Addison-Wesley, Reading, MA
[M+05] Maeda S et al. (2005) A real-time software platform for the cell processor. IEEE Micro 25(5):20–29
[Mil89] Milner R (1989) Communication and Concurrency. Prentice-Hall, Englewood Cliffs
[Pan68] Pankhurst RJ (1968) Program overlay techniques. Communications of the ACM 11(2):119–125
[PvdZ06] Ponse A, van der Zwaag MB (2006) An introduction to program and thread algebra. In: Beckmann A et al (eds) CiE 2006, volume 3988 of Lecture Notes in Computer Science. Springer, Berlin, pp. 445–458
[ST99] Sannella D, Tarlecki A (1999) Algebraic preliminaries. In: Astesiano E, Kreowski H-J, Krieg-Brückner B (eds) Algebraic Foundations of Systems Specification. Springer, Berlin, pp. 13–30
[Stu78] Stubblefield FW (1978) A main program and overlay manager subsystem within a distributed function laboratory computer system. IEEE Trans Nuclear Sci 25(1):217–225
[Wir90] Wirsing M (1990) Algebraic specification. In: van Leeuwen J (ed) Handbook of Theoretical Computer Science, volume B, pp. 675–788. Elsevier, Amsterdam

Received 2 July 2008
Revised 9 November 2010
Accepted 16 March 2011 by C.B. Jones and J.C.P. Woodcock
Published online 21 April 2011