DOI 10.1007/s00165-007-0024-9
BCS © 2007
Formal Aspects of Computing (2007) 19: 445–474

Formal Aspects of Computing

Thread algebra for strategic interleaving

J. A. Bergstra[1,2] and C. A. Middelburg[1,3]

[1]Programming Research Group, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands
[2]Department of Philosophy, Utrecht University, P.O. Box 80126, 3508 TC Utrecht, The Netherlands
[3]Division of Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands.
E-mail: C.A.Middelburg@uva.nl

Abstract. We take a thread as the behavior of a sequential deterministic program under execution and multi-threading as the form of concurrency provided by contemporary programming languages such as Java and C#. We outline an algebraic theory about threads and multi-threading. In the case of multi-threading, some deterministic interleaving strategy determines how threads are interleaved. Interleaving operators for a number of plausible interleaving strategies are specified in a simple and concise way. By that, we show that it is essentially open-ended what counts as an interleaving strategy. We use deadlock freedom as an example to show that there are properties of multi-threaded programs that depend on the interleaving strategy used.

Keywords: Threads; Multi-threading; Thread algebra; Interleaving strategies; Services; Deadlock freedom

1. Introduction

In this paper, we outline an algebraic theory about threads and multi-threading. Theories about concurrent processes such as ACP [BK84, BW90], CCS [Mil80, Mil89], and CSP [BHR84, Hoa85] are based on arbitrary interleaving of parallel processes. However, when dealing with multi-threading, the dominant form of concurrency as provided by recent object-oriented programming languages such as Java [GJSB00] and C# [HWG03], arbitrary interleaving does not provide the most useful intuition. We take the line that, in the case of multi-threading, some deterministic interleaving strategy determines how threads are interleaved. In a relatively simple setting, we define interleaving operators for a number of plausible interleaving strategies. We show further, using deadlock freedom as an example, that there are properties of multi-threaded programs that depend on the interleaving strategy. This provides a way of acquiring an understanding of some essential aspects of multi-threading. Based on the theory developed, we also give an explanation of the kind of synchronization mechanisms on which multi-threading in Java is based, in a way consistent with the most advanced interleaving strategies introduced. Moreover, we outline in brief how the theory developed can be extended such that systems consisting of different multi-threaded programs on the same machine are covered.

Before we discuss our points of departure, it should be clear what a thread is. Attempting to give an informal definition of a thread, we arrive at a listing of key aspects and properties: (i) a thread is the behavior of a sequential deterministic program as run on a machine, (ii) at any time, a thread has some form of unique identity, (iii) a thread will have a time of creation and its individual history thereafter, (iv) during its life a thread co-exists with zero or more other threads in some execution architecture, (v) a thread may interact with system components present in the execution architecture, and (vi) external observations of a thread are made indirectly via the behavior of system components with which the thread interacts.

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

We proceed with discussing our points of departure. They concern: (i) strategic interleaving: the form of interleaving that we consider relevant to multi-threading, (ii) thread vectors: the form taken by the collection of threads in the execution architecture, (iii) services: the kind of system components with which threads interact, and (iv) thread algebra: the theory about threads and multi-threading developed in this paper.

1.1. Strategic interleaving versus arbitrary interleaving

Discrete behaviors, also called processes, proceed by 'doing' steps in a sequential fashion. The simplest view on the parallel composition of two discrete behaviors involves so-called interleaving. In an interleaving, steps of both behaviors occur in some order where each time one step is taken from either the first behavior or the second behavior. Threads will be modeled as discrete behaviors, and each approach to their parallel composition inherits from the theory of parallel composition of discrete behaviors.

Arbitrary interleaving has been proposed by many authors as a plausible, general, if not idealized model of the operation of concurrent systems (see e.g. [Mil80, BK84, BHR84]). Arbitrary interleaving models take into account the totality of all possible interleaving orders in a single model. In the case of arbitrary interleaving, putting two or more processes in parallel results in another process which incorporates all conceivable ways in which the steps of the given processes can be interleaved. Each different mathematical representation of processes induces its own way of obtaining the parallel composition of a number of processes in an arbitrary interleaving fashion. One might say that the concept of arbitrary interleaving depends upon the mathematical representation of processes under consideration.

In contrast to arbitrary interleaving concurrency, so-called true concurrency models have been proposed which, in some cases, offer a better understanding of parallelism (see e.g. [Pet62, Rei85]). In true concurrency models, one intends to avoid causal dependencies that may solely arise from the sequencing of steps given by any arbitrary interleaving of two processes. There are numerous true concurrency models just as there is a multitude of arbitrary interleaving concurrency models. For the theory and practice of computer programming, however, another contrast with arbitrary interleaving comes to mind.

Rather than assuming some mechanism of arbitrary interleaving it may be assumed that some deterministic interleaving strategy determines the ordering of steps of various processes – as is the case with the form of concurrency provided by C# and Java. This strategy need not be known to a programmer who may be happy to know that some strategy chosen from a collection of adequate strategies is applied. We propose to use the phrase *strategic interleaving* for the more constrained form of interleaving obtained by using such a strategy. We consider strategic interleaving to be the form of interleaving that is relevant to multi-threading.

Arbitrary interleaving and strategic interleaving are quite different. It happens that interleaving of certain threads leads to deadlock with a particular deterministic interleaving strategy whereas arbitrary interleaving would not lead to deadlock, and vice versa. This illustrates why arbitrary interleaving does not provide an appropriate abstraction when dealing with multi-threading. True concurrency and strategic interleaving are completely different. True concurrency is unfit for dealing with multi-threading because it is a non-interleaving form of concurrency.

1.2. Thread algebra versus process algebra

Thread algebra is an algebraic theory of processes which is focused on strategic interleaving. Process algebra, in contrast, has the much more general purpose of providing a general theory of parallel composition of systems. Process algebra is designed to have parallel composition operators with fundamental properties such as commutativity and associativity, which are not considered essential in thread algebra due to the 'axiom' that in a multi-threaded execution architecture each thread has its unique place, although at some level of abstraction almost nothing may be known about that place. In this paper, where the collection of threads is considered to take the form of a sequence, that place will be the position of the thread in the sequence.

Thread algebra[1] is developed by extending BPPA (Basic Polarized Process Algebra) [BL02, BB03]. BPPA is far less general than ACP-style process algebras and its design focuses on the semantics of deterministic sequential

[1] The phrase 'thread algebra' can be found on the web as proposed by James Orsilio around 1990. It has served as the theoretical basis of a software product of Orthstar. No information regarding its mathematical content seems to have been published, however. No other occurrences of this phrase have been observed by the authors.

programs. The semantics of a deterministic sequential program is supposed to be a polarized process. The idea is that a polarized process may occur in two roles: the role of a client and the role of a server. In the former role, basic actions performed by the polarized process are requests upon which a reply is expected. In the latter role, basic actions performed by the polarized process are offers to serve a request and to return a reply. The distinction between these roles is relevant in case BPPA is extended with a mechanism for client-server interaction, as in [BB05]. However, BPPA itself deals with polarized processes that occur in the role of a client only. In thread algebra, seeing that threads are taken for the behaviors of deterministic sequential programs under execution, all threads are viewed as polarized processes that occur in the role of a client only.

One of the assumptions made in thread algebra is that a deterministic interleaving strategy determines how threads are interleaved. The exclusion of non-deterministic interleaving strategies is a simplification. We believe however that some simplifications are needed to obtain a manageable theory about threads and multi-threading. It is obvious that the resulting theory will be inadequate in certain cases, for instance, in the case where the interleaving strategy requires a randomizer.

1.3. A taxonomy of services

The well-known client-server dichotomy fails to provide the terminology that we need for the present purposes. Taking the thread as the particular kind of client under consideration, a phrase is required to indicate the system part to which the commands of a thread might be directed. In [BP02], the dichotomy has been given as program versus state machine. There, the program represents a thread, though less abstract, and a state machine is a reactive component to which a program issues its instructions. This suggests that threads coordinate the activity of reactive components. So we could propose 'reactor' as a technical term for a system component to which a thread issues it commands. A disadvantage of the term reactor, however, is that it fails to have a meaning independent of the concept of an actor. In addition, a thread may issue its commands to an active component just as well.

Looking for an abstract term or phrase that instantiates the concept of a component accepting commands issued by a thread, and has some independent significance as well, we arrive at 'service'. Thus, a thread issues its commands, in the sequel called basic actions, to one or more services. Services can be specified and analyzed in a service algebra, which is of no concern to us here, however. In the setting of multi-threading, services may be classified in several ways. A major distinction is between target services and para-target services; another distinction is between shared services and local services:

target services. A target service processes commands in a context observable by external parties. Printing a document, sending an email message or showing data on a display are typical examples of calling a target service. Writing persistent data in permanent memory is another example, because permanent memory may be read by other running programs, in particular programs that start execution after the writing thread has terminated. The very reason for any collection of threads to be run always resides in the collective effect that all commands involved have on the target services provided by the execution architecture.

para-target services. Services that are not target services will be called para-target services. Alternative names are auxiliary services, internal services and so on. We propose to use the phrase para-target in order to avoid any misleading connotations. Transferring data by means of a Java pipe or setting a timer are typical examples of calling a para-target service. A para-target service cannot store persistent data.

local services. A local service is accessible to a single thread only. If it has a state, that state is initialized when the thread is created and its state is under the complete control of that single thread. Local services can be identified as either local target services or local para-target services. A timer is usually a local para-target service. Local para-target services exist to support a thread in creating useful, or at least intended, behavior towards target services.

shared services. A shared service in a multi-threaded system provides its service to all existing threads. Shared services, like local services, can be identified as either shared target services or shared para-target services. A Java pipe is an example of a shared para-target service.

To simplify the setting, it will be assumed that all services are either local or shared, thus disregarding the possibility that a service is accessible to some class of threads only, and also that there are no local target services. This leaves us with execution architectures that provide shared target services, shared para-target services and local para-target services.

The overall intuition about threads, target services and para-target services is that:

- a polarized process is the behavior of a sequential deterministic program;
- a multi-threaded system consists of a number of polarized processes, called threads, which are interleaved according to some deterministic interleaving strategy and which interact with a number of services;
- the intentions about the resulting behavior pertain only to interaction with target services;
- interaction with para-target services takes place only in as far as it is needed to obtain the intended behavior in relation to target services.

One of the assumptions made in thread algebra is that para-target services are deterministic. The exclusion of non-deterministic para-target services, like the exclusion of non-deterministic interleaving strategies, is a simplification. We believe however that this simplification is adequate in the cases that we address: services that keep local data for a thread or shared data for a number of threads. Of course, it is inadequate in cases where services such as dice-playing services are taken into consideration. Another assumption is that target services are non-deterministic. The reason for this assumption is that the dependence of target services on external conditions make it appear to threads that they behave non-deterministically.

1.4. Thread vectors and strategic interleaving operators

In order to deal with multi-threading, it is assumed that a collection of threads to be interleaved takes the form of a sequence, called a thread vector. Strategic interleaving operators describe interleaving strategies on thread vectors. These operators are the content of thread algebra. There are various alternatives for the assumption that a collection of threads to be interleaved is a thread vector. The assumption made in thread algebra results in a relatively simple setting. As explained below, it has essentially the same purpose as the frequently occurring assumption that the pool of programming variables of a program takes the form of a sequence.

By limiting the discussion to the relatively simple setting of thread algebra, only a restricted number of strategic interleaving operators can be modeled. The key advantage, however, is that interleaving strategies characteristic of multi-threading can be specified in a concise and comprehensible way. The family of strategic interleaving operators grows by introducing more features for thread behavior and interaction, the simplest case being cyclic interleaving. Features treated in this paper include thread creation, thread termination, enabledness tests, and locking/unlocking of services. By designing an incremental hierarchy of strategic interleaving operators, complicated program notation designs can be understood by indicating what might be a plausible implementation for some set of multi-threading features.[2]

1.5. Outline of the paper

The structure of this paper is as follows. After a review of BPPA (Sect. 2), we extend it with various deterministic interleaving strategies on thread vectors. First, we introduce interleaving strategies that do not take into account execution architectures in which there are actions that cannot always be performed (Sect. 3). Next, we introduce interleaving strategies that can deal with such blocking actions (Sect. 4). Following this, using examples in which blocking actions are involved, we show by informal reasoning that it may depend on the ordering of the threads in the thread vector or the interleaving strategy used whether the interleaving of a thread vector leads to deadlock (Sect. 5). After a recapitulation of earlier remarks about execution architectures (Sect. 6), we make precise what it means that the interleaving of a thread vector leads to deadlock in some context of para-target services (Sect. 7). Then, we give an explanation of the kind of synchronization mechanisms on which multi-threading in Java is based, in a way consistent with the interleaving strategies that can deal with blocking actions (Sect. 8), and outline in brief how thread algebra can be extended such that systems consisting of different multi-threaded programs on the same machine are covered (Sect. 9). Finally, we make some concluding remarks (Sect. 10).

[2] We feel that this kind of description is hardly doable in ordinary arbitrary interleaving based process algebra, mainly because the necessity to maintain a commutative and associative parallel composition forces one into a use of choice which is not commonly implied in the use of multi-threading. The use of states and guards is also more easily supported in thread algebra.

Table 1. Axiom of BPPA

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

2. Basic polarized process algebra

In this section, we review BPPA (Basic Polarized Process Algebra), a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution.

2.1. Constants, operators and axioms

In BPPA, it is assumed that there is a fixed but arbitrary finite set $\mathcal{A}$ of *basic actions* with $\mathsf{tau} \notin \mathcal{A}$. We write $\mathcal{A}_{\mathsf{tau}}$ for $\mathcal{A} \cup \{\mathsf{tau}\}$. BPPA has the following constants and operators:

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

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

The intuition is that each basic action is taken as a command to be processed by the execution environment. More specifically, a basic action is taken as a command for a service offered by the execution environment. 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 concerned produces a reply value. This reply is either T or F and is returned to the polarized process issuing the command. The polarized process $x \trianglelefteq a \trianglerighteq y$ will proceed as x if the processing of a leads to the reply T (called a positive reply), and it will proceed as y if the processing of a leads to the reply F (called a negative reply). If the reply is used to indicate whether the processing was successful, a useful convention is to indicate successful processing by the reply T and unsuccessful processing by the reply F. The action tau plays a special role. Its execution will never change any state and always produces a positive reply. It is a concrete internal action. This means that its presence matters, and that no abstraction is made of it. The name tau is taken from the CCS notation for silent step in (non-polarized) process algebra, but the Greek letter is not used here because the characteristic equations of such silent steps are not implied. BPPA has only one axiom. This axiom is given in Table 1. Using the abbreviation introduced above, it can be written as follows: $x \trianglelefteq \mathsf{tau} \trianglerighteq y = \mathsf{tau} \circ x$.

2.2. Guarded recursion and the approximation induction principle

A *system of recursion equations* over BPPA is a set of equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables and each t_X is a term of BPPA that only contains variables from V. Let t be a term of BPPA containing a variable X. Then an occurrence of X in t is *guarded* if t has a subterm of the form $t' \trianglelefteq a \trianglerighteq t''$ containing this occurrence of X. A system of recursion equations E is *guarded* if all occurrences of variables in the right-hand sides of its equations are guarded or it can be rewritten to such a system of recursion equations using the equations of E. We are only interested in models of BPPA in which guarded systems of recursion equations have unique solutions, such as the projective limit model of BPPA presented in [BB03, BL02].

The projective limit model is based on the notion of a finite approximation of depth n. When for all n these approximations are identical for two given polarized processes, both polarized processes are considered identical. This is expressed by the infinitary conditional equation AIP (Approximation Induction Principle) given in Table 2. Following [BB03, BL02], approximation of depth n is phrased in terms of a unary *projection* operator $\pi_n(\_)$. The projection operators are defined inductively by means of the axioms P0–P3 given in Table 2. In P3, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. Axioms P0–P3 express that the n-th projection of a polarized process is the same as the polarized process itself except that it deadlocks after being executed up to depth n. The projection operators defined here are reminiscent of the projection operators that were added to ACP in [vG87].

Table 2. Approximation induction principle

$\pi_0(x) = \mathsf{D}$	P0
$\pi_{n+1}(\mathsf{S}) = \mathsf{S}$	P1
$\pi_{n+1}(\mathsf{D}) = \mathsf{D}$	P2
$\pi_{n+1}(x \trianglelefteq a \trianglerighteq y) = \pi_n(x) \trianglelefteq a \trianglerighteq \pi_n(y)$	P3
$(\bigwedge_{n \geq 0} \pi_n(x) = \pi_n(y)) \Rightarrow x = y$	AIP

2.3. Structural operational semantics

As mentioned above, the behavior of a polarized process depends upon its execution environment. Each action performed by the polarized process is taken as a command to be processed by the execution environment. At any stage, the commands that the execution environment can accept depend only on its history, i.e. the sequence of commands processed before and the sequence of replies produced for those commands. When the execution environment accepts a command, it will produce a positive reply or a negative reply. Whether the reply is positive or negative usually depends on the execution history. However, it may also depend on external conditions. For example, when the execution environment accepts a command to write a file to a diskette, it will usually produce a positive reply, but not if the diskette turns out to be write-protected. Ignorance of its dependence on external conditions makes it appear that an execution environment behaves non-deterministically. The sketched operation of an execution environment leads us to represent it as described below.

In the structural operational semantics of BPPA, an execution environment is represented by a function $\rho : (\mathcal{A} \times \{\mathsf{T}, \mathsf{F}\})^* \to \mathcal{P}(\mathcal{A} \times \{\mathsf{T}, \mathsf{F}\})$ that satisfies the condition $(a, b) \notin \rho(\alpha) \Rightarrow \rho(\alpha \frown \langle (a, b) \rangle) = \emptyset$ for all $a \in \mathcal{A}$, $b \in \{\mathsf{T}, \mathsf{F}\}$, and $\alpha \in (\mathcal{A} \times \{\mathsf{T}, \mathsf{F}\})^*$.[3] We write $\mathcal{E}$ for the set of all those functions. Given an execution environment $\rho \in \mathcal{E}$ and an action $a \in \mathcal{A}$, the *derived* execution environment of ρ after processing a with a *positive* reply, written $\frac{\partial^+}{\partial a}\rho$, is defined by $\frac{\partial^+}{\partial a}\rho(\alpha) = \rho(\langle (a, \mathsf{T}) \rangle \frown \alpha)$; and the *derived* execution environment of ρ after processing a with a *negative* reply, written $\frac{\partial^-}{\partial a}\rho$, is defined by $\frac{\partial^-}{\partial a}\rho(\alpha) = \rho(\langle (a, \mathsf{F}) \rangle \frown \alpha)$.

The following transition relations on closed terms are used in the structural operational semantics of BPPA:

- a binary relation $\langle \_, \rho \rangle \xrightarrow{a} \langle \_, \rho' \rangle$ for each $a \in \mathcal{A}_{\mathsf{tau}}$ and $\rho, \rho' \in \mathcal{E}$;
- a unary relation $\_\downarrow$;
- a unary relation $\_\uparrow$;
- a unary relation $\_\updownarrow$.

The four kinds of transition relations are called the *action step*, *termination*, *deadlock*, and *termination or deadlock* relations, respectively. They can be explained as follows:

- $\langle t, \rho \rangle \xrightarrow{a} \langle t', \rho' \rangle$: in execution environment ρ, process t can perform action a and after that proceed as process t' in execution environment ρ';
- $t\downarrow$: process t cannot but terminate successfully;
- $t\uparrow$: process t cannot but become inactive;
- $t\updownarrow$: process t cannot but terminate successfully or become inactive.

The termination or deadlock relation is an auxiliary relation needed when we extend BPPA with strategic interleaving operators.

The structural operational semantics of BPPA is described by the transition rules given in Table 3 where a ranges over $\mathcal{A}$. The structural operational semantics for the operators $\pi_n(\_)$ is described by the transition rules given in Table 4 where a ranges over $\mathcal{A}_{\mathsf{tau}}$.

In the presence of recursion, the structural operational semantics needs a special provision, namely constants for the solutions of guarded systems of recursion equations. We add to the constants of BPPA, for each guarded

[3] We write $\langle\,\rangle$ for the empty sequence, $\langle d \rangle$ for the sequence having d as sole element, and $\alpha \frown \beta$ for the concatenation of finite sequences α and β. We assume the usual laws for concatenation of finite sequences. 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.

Table 3. Structural operational semantics of BPPA

$$\frac{}{\mathsf{S}\downarrow} \qquad \frac{}{\mathsf{D}\uparrow} \qquad \frac{}{\langle x \trianglelefteq \mathsf{tau} \trianglerighteq y, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x, \rho\rangle}$$

$$\frac{}{\langle x \trianglelefteq a \trianglerighteq y, \rho\rangle \xrightarrow{a} \langle x, \frac{\partial^+}{\partial a}\rho\rangle}\ (a, \mathsf{T}) \in \rho(\langle\,\rangle) \qquad \frac{}{\langle x \trianglelefteq a \trianglerighteq y, \rho\rangle \xrightarrow{a} \langle y, \frac{\partial^-}{\partial a}\rho\rangle}\ (a, \mathsf{F}) \in \rho(\langle\,\rangle)$$

$$\frac{x\downarrow}{x\updownarrow} \qquad \frac{x\uparrow}{x\updownarrow}$$

Table 4. Structural operational semantics for projection

$$\frac{\langle x, \rho\rangle \xrightarrow{a} \langle x', \rho'\rangle}{\langle \pi_{n+1}(x), \rho\rangle \xrightarrow{a} \langle \pi_n(x'), \rho'\rangle} \qquad \frac{x\downarrow}{\pi_{n+1}(x)\downarrow} \qquad \frac{x\uparrow}{\pi_{n+1}(x)\uparrow} \qquad \frac{}{\pi_0(x)\uparrow}$$

system of recursion equations E and each variable X that occurs as the left-hand side of an equation in E, a constant standing for the unique solution of E for X. This constant is denoted by $\langle X|E\rangle$. The structural operational semantics for the constants $\langle X|E\rangle$ is described by the transition rules given in Table 5 where a ranges over $\mathcal{A}_{\mathsf{tau}}$. Here we write $\langle t|E\rangle$ for t with, for all X that occur on the left-hand side of an equation in E, all occurrences of X in t replaced by $\langle X|E\rangle$.

Bisimulation equivalence is defined as follows. A *bisimulation* is a symmetric binary relation B on closed terms such that for all closed terms t_1 and t_2:

- if $B(t_1, t_2)$ and $\langle t_1, \rho\rangle \xrightarrow{a} \langle t_1', \rho'\rangle$, then there is a t_2' such that $\langle t_2, \rho\rangle \xrightarrow{a} \langle t_2', \rho'\rangle$ and $B(t_1', t_2')$;
- if $B(t_1, t_2)$ and $t_1\downarrow$, then $t_2\downarrow$;
- if $B(t_1, t_2)$ and $t_1\uparrow$, then $t_2\uparrow$.

Two closed terms t_1 and t_2 are *bisimulation equivalent*, written $t_1 \leftrightarrows t_2$, if there exists a bisimulation B such that $B(t_1, t_2)$.

Bisimulation equivalence is a congruence. This follows immediately from the fact that the transition rules for BPPA constitute a transition system specification in path format (see e.g. [AFV01]). The axioms given in Tables 1 and 2 are sound with respect to bisimulation equivalence.

Pairs consisting of a closed term and an execution environment are sometimes called *configurations*. A variant of bisimulation equivalence that is coarser could be obtained by relating configurations instead of terms. However, different from bisimulation equivalence as defined above, that variant would not even be a congruence with respect to the simplest strategic interleaving operator added to BPPA in this paper. In the terminology of [MRG05], bisimulation equivalence as defined above is *stateless* bisimulation equivalence and the intended variant is *initially stateless* bisimulation equivalence.

2.4. Conditionals

Henceforth, use is made of the auxiliary conditional operator $\_ \trianglelefteq \_ \trianglerighteq \_$, where the second argument must equal T or F. The conditional operator is defined by means of the axioms in Table 6. The conditional operator defined here has been taken from [BB92], where it was introduced in the setting of ACP. The conditional operator is

Table 5. Structural operational semantics for guarded recursion

$$\frac{\langle\langle t|E\rangle, \rho\rangle \xrightarrow{a} \langle x', \rho'\rangle}{\langle\langle X|E\rangle, \rho\rangle \xrightarrow{a} \langle x', \rho'\rangle}\ X = t \in E \qquad \frac{\langle t|E\rangle\downarrow}{\langle X|E\rangle\downarrow}\ X = t \in E \qquad \frac{\langle t|E\rangle\uparrow}{\langle X|E\rangle\uparrow}\ X = t \in E$$

Table 6. Axioms for conditional operator

$x \trianglelefteq \mathsf{T} \trianglerighteq y = x$	CO1
$x \trianglelefteq \mathsf{F} \trianglerighteq y = y$	CO2

not considered part of BPPA. It is added to BPPA here because it happens to be convenient in defining various strategic interleaving operators.

3. Interleaving strategies for non-blocking actions

In [BL02], it has been outlined how and why polarized processes are a natural candidate for the description of sequential program semantics. Assuming that a thread is a process representing a program under execution, it is reasonable to view all polarized processes as threads. From that point of view, BPPA is an algebraic theory about threads. In this section, we take up the extension of this theory to a theory about threads and multi-threading by introducing various simple interleaving strategies. We first treat a very simple strategy, to wit (pure) cyclic interleaving. Next, we treat a strategy based on cyclic interleaving where basic actions from different threads can be performed simultaneously and a strategy based on cyclic interleaving where each thread is given a fixed number of consecutive turns. We also treat a strategy not based on cyclic interleaving. After that, we treat several strategies based on cyclic interleaving that support thread creation. Finally, we treat a strategy based on cyclic interleaving that supports both thread creation and thread termination. The simple strategies introduced in this section do not take into account execution architectures in which there are actions, other than thread creation actions, that cannot always be performed. Interleaving strategies that can deal with such blocking actions will be introduced in Sect. 4.

As mentioned in Sect. 1, it is assumed that a collection of threads to be interleaved takes the form of a sequence of threads, called a *thread vector*. Strategic interleaving operators turn a thread vector of arbitrary length into a single thread. This single thread obtained via a strategic interleaving operator is also called a multi-thread. Formally, however, both threads and multi-threads are polarized processes and there is no further difference in type.

Subscripts of strategic interleaving operators in lower case will be used to indicate features which are dealt with in addition to the minimum given by the strategic interleaving operator for cyclic interleaving introduced first. Subscripts of strategic interleaving operators in upper case will be used to indicate interleaving strategies which are not based on cyclic interleaving. Superscripts will be used to encode state information when needed. In the report version of this paper, every strategic interleaving operator based on cyclic interleaving has a subscript starting with csi, standing for cyclic strategic interleaving, but this convention turned out to be unhandy.

3.1. Cyclic interleaving

We first introduce the strategic interleaving operator for cyclic interleaving. This operator basically operates as follows: at each interleaving step, the first thread in the thread vector gets a turn to perform an action and then the remaining 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. The axioms for cyclic interleaving ($\|(\_)$) are given in Table 7. In CSI4, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. In CSI3, the auxiliary *deadlock at termination* operator $\mathsf{S}_{\mathsf{D}}(\_)$ is used. This operator turns termination into deadlock. Its axioms appear in Table 8. In S2D3, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$.

The axioms for cyclic interleaving constitute a definition by recursion on the sum of the depths of all threads in the thread vector with case distinction on the structure of the first thread in the thread vector. Hence, it is obvious that the axioms are consistent and that every closed term of BPPA extended with cyclic interleaving is derivably equal to a closed term of BPPA. Moreover, it follows easily that every projection of a closed term of BPPA extended with guarded recursion and cyclic interleaving is derivably equal to a closed term of BPPA. Similar remarks apply to all strategic interleaving operators introduced in this paper and will not be repeated.

Table 7. Axioms for cyclic interleaving

$\|(\langle\rangle) = \mathsf{S}$	CSI1
$\|(\langle \mathsf{S}\rangle \frown \alpha) = \|(\alpha)$	CSI2
$\|(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_\mathsf{D}(\|(\alpha))$	CSI3
$\|(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) = \|(\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|(\alpha \frown \langle y\rangle)$	CSI4

Table 8. Axioms for deadlock at termination

$\mathsf{S}_\mathsf{D}(\mathsf{S}) = \mathsf{D}$	S2D1
$\mathsf{S}_\mathsf{D}(\mathsf{D}) = \mathsf{D}$	S2D2
$\mathsf{S}_\mathsf{D}(x \trianglelefteq a \trianglerighteq y) = \mathsf{S}_\mathsf{D}(x) \trianglelefteq a \trianglerighteq \mathsf{S}_\mathsf{D}(y)$	S2D3

The structural operational semantics for the operators $\|(\_)$ and $\mathsf{S}_\mathsf{D}(\_)$ is described by the transition rules given in Table 9 where a ranges over $\mathcal{A}_\mathsf{tau}$.

Bisimulation equivalence is also a congruence with respect to the operators $\|(\_)$ and $\mathsf{S}_\mathsf{D}(\_)$. This follows immediately from the fact that the transition rules for BPPA extended with these operators constitute a complete transition system specification in relaxed panth format (see e.g. [Mid03]). The axioms given in Tables 7 and 8 are sound with respect to bisimulation equivalence.

Structural operational semantics can also be given for each of the other strategic interleaving operators treated in this paper. We will refrain from doing so. For those other strategic interleaving operators, the description of the structural operational semantics is similar, but more involved, and it adds at most marginally to a better understanding of the interleaving strategy concerned.

3.2. Basic action width two and beyond

Cyclic interleaving as introduced in Sect. 3.1 excludes basic actions from different threads from being performed simultaneously. The number of basic actions that can be performed simultaneously is called the basic action width. In these terms, $\|(\_)$ provides basic action width one only. Basic actions a and b are independent, written $a \# b$, if both can be performed simultaneously with an effect that equals the effect of performing them in either of the two possible orderings. The result of performing independent actions a and b simultaneously is considered to be a basic action, which is denoted by $a \mid b$.

Assuming that independence is known as a relation given on basic actions, a strategic interleaving operator may issue $a \mid b$ whenever possible. Simultaneous basic action issuing is vital for so-called micro-threads which are used to speed up processors by maximizing execution width (see e.g. [JL00]). In order to define a strategic interleaving operator for basic action width 2, it is a reasonable simplification to assume that basic actions independent of other basic actions always produce a positive reply, and that the simultaneous execution of two

Table 9. Structural operational semantics for cyclic interleaving and deadlock at termination

$$\frac{x_1\downarrow, \ldots, x_k\downarrow, \langle x_{k+1}, \rho\rangle \xrightarrow{a} \langle x'_{k+1}, \rho'\rangle}{\langle \|(\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha), \rho\rangle \xrightarrow{a} \langle \|(\alpha \frown \langle x'_{k+1}\rangle), \rho'\rangle} \quad (k \geq 0)$$

$$\frac{x_1\updownarrow, \ldots, x_k\updownarrow, x_l\uparrow, \langle x_{k+1}, \rho\rangle \xrightarrow{a} \langle x'_{k+1}, \rho'\rangle}{\langle \|(\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha), \rho\rangle \xrightarrow{a} \langle \|(\alpha \frown \langle \mathsf{D}\rangle \frown \langle x'_{k+1}\rangle), \rho'\rangle} \quad (k \geq l > 0)$$

$$\frac{x_1\downarrow, \ldots, x_k\downarrow}{\|(\langle x_1\rangle \frown \ldots \frown \langle x_k\rangle)\downarrow} \quad (k \geq 0) \qquad \frac{x_1\updownarrow, \ldots, x_k\updownarrow, x_l\uparrow}{\|(\langle x_1\rangle \frown \ldots \frown \langle x_k\rangle)\uparrow} \quad (k \geq l > 0)$$

$$\frac{\langle x, \rho\rangle \xrightarrow{a} \langle x', \rho'\rangle}{\langle \mathsf{S}_\mathsf{D}(x), \rho\rangle \xrightarrow{a} \langle \mathsf{S}_\mathsf{D}(x'), \rho'\rangle} \qquad \frac{x\updownarrow}{\mathsf{S}_\mathsf{D}(x)\uparrow}$$

Table 10. Axioms for cyclic interleaving with basic action width 2

$\|_{w2}(\langle\rangle) = \mathsf{S}$	CSIw1
$\|_{w2}(\langle \mathsf{S}\rangle \frown \alpha) = \|_{w2}(\alpha)$	CSIw2
$\|_{w2}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_\mathsf{D}(\|_{w2}(\alpha))$	CSIw3
$\|_{w2}(\langle x \trianglelefteq a \trianglerighteq y\rangle) = \|_{w2}(\langle x\rangle) \trianglelefteq a \trianglerighteq \|_{w2}(\langle y\rangle)$	CSIw4
$\|_{w2}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \langle u \trianglelefteq b \trianglerighteq v\rangle \frown \alpha) =$ $(a \mid b \circ \|_{w2}(\alpha \frown \langle x\rangle \frown \langle u\rangle)) \lhd a\#b \rhd (\|_{w2}(\langle u \trianglelefteq b \trianglerighteq v\rangle \frown \alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|_{w2}(\langle u \trianglelefteq b \trianglerighteq v\rangle \frown \alpha \frown \langle y\rangle))$	CSIw5

Table 11. Axioms for cyclic interleaving with step counting

$\|^k(x) = \|^{k,1}(x)$	CSIsc0
$\|^{k,l}(\langle\rangle) = \mathsf{S}$	CSIsc1
$\|^{k,l}(\langle \mathsf{S}\rangle \frown \alpha) = \|^{k,1}(\alpha)$	CSIsc2
$\|^{k,l}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_\mathsf{D}(\|^{k,1}(\alpha))$	CSIsc3
$\|^{k,l}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) = (\|^{k,1}(\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|^{k,1}(\alpha \frown \langle y\rangle)) \lhd k = l \rhd (\|^{k,l+1}(\langle x\rangle \frown \alpha) \trianglelefteq a \trianglerighteq \|^{k,l+1}(\langle y\rangle \frown \alpha))$	CSIsc4

independent basic actions always yields a positive reply. Thus, if a basic action may produce both a positive reply and a negative reply, it cannot be performed simultaneously with any other basic action. A strategy with basic action width 2 is presented in Table 10. In CSIw4 and CSIw5, a and b stand for arbitrary actions from $\mathcal{A}_{\mathsf{tau}}$. A similar but more complicated axiomatization can be found for higher basic action widths of course. The remainder of this paper focuses on the case of basic action width one which is vital for an understanding of multi-thread computer programming, leaving for future elaboration a development in the direction of processor architecture.

3.3. Step counting

A simple variation of cyclic interleaving is cyclic interleaving with step counting ($\|^{k,l}(\_)$), which is equipped with a counter and gives each thread a fixed number k of consecutive turns. The counter l indicates that $l-1$ of the k steps have already been performed ($1 \leq l \leq k$). The axioms for cyclic interleaving with step counting are given in Table 11. In CSIsc4, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. CSIsc0 defines an additional operator: $\|^k(\_)$. Clearly, for all x, $\|(x) = \|^1(x)$. The advantage of this interleaving strategy is that fewer context switches, i.e. moves from one thread to another one, are made, which may in some cases speed up execution.

Yielding

Yielding within a thread stands for handing over control to another thread. This becomes meaningful in the step counting strategy with $k > 1$. Step counting starts over again for the other thread. Yielding has no effect if there is no other thread in the thread vector. In Table 12, an axiom for yielding is given. Recall that the special action tau is an internal action. Here, it arises as the residue of yielding.

3.4. Current thread persistence

Having available the action YIELD, or any other action that invokes permutation of the thread vector, cyclic interleaving may be dropped in favor of permutations explicitly asked for by means of the action YIELD. A strategy of this kind is said to provide current thread persistence, thus expressing that the current thread switches to

Table 12. Additional axiom for cyclic interleaving with step counting and yielding

$\|^{k,l}(\langle x \trianglelefteq \mathsf{YIELD} \trianglerighteq y\rangle \frown \alpha) = \mathsf{tau} \circ (\|^{k,1}(\alpha \frown \langle x\rangle) \lhd \alpha \neq \langle\rangle \rhd \|^{k,l+1}(\langle y\rangle))$	CSIscY

Table 13. Axioms for a strategy with current thread persistence

$\Vert_{\mathrm{CTP}}(\langle\rangle) = \mathsf{S}$	ctpSI1
$\Vert_{\mathrm{CTP}}(\langle \mathsf{S}\rangle \frown \alpha) = \Vert_{\mathrm{CTP}}(\alpha)$	ctpSI2
$\Vert_{\mathrm{CTP}}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\Vert_{\mathrm{CTP}}(\alpha))$	ctpSI3
$\Vert_{\mathrm{CTP}}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) = \Vert_{\mathrm{CTP}}(\langle x\rangle \frown \alpha) \trianglelefteq a \trianglerighteq \Vert_{\mathrm{CTP}}(\langle y\rangle \frown \alpha)$	ctpSI4
$\Vert_{\mathrm{CTP}}(\langle x \trianglelefteq \mathsf{YIELD} \trianglerighteq y\rangle \frown \alpha) = \mathsf{tau} \circ (\Vert_{\mathrm{CTP}}(\alpha \frown \langle x\rangle) \lhd \alpha \neq \langle\rangle \rhd \Vert_{\mathrm{CTP}}(\langle y\rangle))$	ctpSIY

Table 14. Additional axiom for projection

$\pi_{n+1}(x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y) = \pi_n(x) \trianglelefteq \mathsf{NT}(\pi_n(z)) \trianglerighteq \pi_n(y)$	PNT

another thread only when the current thread explicitly asks for it. The family of strategies that is outlined below will be based on cyclic interleaving rather than on current thread persistence because some form of permutation taking place outside the control of the individual threads is considered essential for multi-threading at any rate. Table 13 provides axioms for a strategy with current thread persistence where the thread vector undergoes cyclic permutation when the current thread explicitly hands over control to another thread. In ctpSI4, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$ different from the action YIELD.

3.5. Thread creation or forking

Forking off a thread is a step of some thread which gives rise to the creation of a new additional thread which will be running in the same context. We intend to separate the act of forking off the thread from the interleaving strategy subsequently dealing with the new thread. In particular, a fork action may succeed, giving rise to a new thread indeed, or fail, in which case no new thread is created. The fork action produces a positive reply if and only if it succeeds. This allows the thread performing the fork action to make its further behavior dependent on whether or not the fork action actually succeeded. The success may depend on a variety of aspects which are immaterial to the act of forking off as such. In order to formalize these intuitions, an operator 'new thread' ($\mathsf{NT}(x)$) is introduced which represents the act of trying to fork off a thread x. Thus, $\mathsf{NT}(x)$ is viewed as a basic action ignoring the way the new thread may be dealt with by an interleaving strategy.

The thread forking operator may appear inside guarded systems of recursion equations. To deal with that matter, the projective limit model characterization of process identity on polarized processes, which easily carries over to this case, is used. The projection operators are extended inductively by means of the axiom in Table 14. The working of AIP from Table 2 in this case can be appreciated when considering a system of recursion equations such as:

$$P = R \trianglelefteq a \trianglerighteq \mathsf{S}\,,$$
$$Q = P \trianglelefteq \mathsf{NT}(R \trianglelefteq a \trianglerighteq Q) \trianglerighteq \mathsf{D}\,,$$
$$R = Q \trianglelefteq d \trianglerighteq e \circ \mathsf{S}\,.$$

With AIP, $\pi_n(P)$ is provably equal to a closed term for each n.

The axioms for cyclic interleaving with forking ($\Vert_{\mathrm{f}}(\_)$) are given in Table 15. In CSIf4, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$ different from the actions of the form $\mathsf{NT}(x)$ and the action NT. In CSIf5, the additional basic action NT is used. It produces a positive reply if the creation of a new thread can take place and it produces a negative reply if the creation of a new thread cannot take place. For instance, the execution architecture may impose a maximum on the length of the thread vector. In that case, NT will produce a positive reply only if the length of the thread vector is still below the maximum.

3.6. Thread forking combined with step counting

Features can be combined by integrating equation systems for different interleaving strategies. Table 16 shows the result of combining in a single strategic interleaving operator both step counting (without yielding) and forking.

Table 15. Axioms for cyclic interleaving with forking

$\lVert_{\mathrm{f}}(\langle\rangle) = \mathsf{S}$	CSIf1
$\lVert_{\mathrm{f}}(\langle \mathsf{S}\rangle \frown \alpha) = \lVert_{\mathrm{f}}(\alpha)$	CSIf2
$\lVert_{\mathrm{f}}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\lVert_{\mathrm{f}}(\alpha))$	CSIf3
$\lVert_{\mathrm{f}}(\langle x \trianglelefteq\ a\ \trianglerighteq y\rangle \frown \alpha) = \lVert_{\mathrm{f}}(\alpha \frown \langle x\rangle) \trianglelefteq\ a\ \trianglerighteq \lVert_{\mathrm{f}}(\alpha \frown \langle y\rangle)$	CSIf4
$\lVert_{\mathrm{f}}(\langle x \trianglelefteq\ \mathsf{NT}(z)\ \trianglerighteq y\rangle \frown \alpha) = \lVert_{\mathrm{f}}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq\ \mathsf{NT}\ \trianglerighteq \lVert_{\mathrm{f}}(\alpha \frown \langle y\rangle)$	CSIf5

Table 16. Axioms for cyclic interleaving with step counting and forking

$\lVert_{\mathrm{f}}^{k}(x) = \lVert_{\mathrm{f}}^{k,l}(x)$	CSIscf0
$\lVert_{\mathrm{f}}^{k,l}(\langle\rangle) = \mathsf{S}$	CSIscf1
$\lVert_{\mathrm{f}}^{k,l}(\langle \mathsf{S}\rangle \frown \alpha) = \lVert_{\mathrm{f}}^{k,1}(\alpha)$	CSIscf2
$\lVert_{\mathrm{f}}^{k,l}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\lVert_{\mathrm{f}}^{k,1}(\alpha))$	CSIscf3
$\lVert_{\mathrm{f}}^{k,l}(\langle x \trianglelefteq\ a\ \trianglerighteq y\rangle \frown \alpha) = (\lVert_{\mathrm{f}}^{k,1}(\alpha \frown \langle x\rangle) \trianglelefteq\ a\ \trianglerighteq \lVert_{\mathrm{f}}^{k,1}(\alpha \frown \langle y\rangle)) \lhd k = l \rhd (\lVert_{\mathrm{f}}^{k,l+1}(\langle x\rangle \frown \alpha) \trianglelefteq\ a\ \trianglerighteq \lVert_{\mathrm{f}}^{k,l+1}(\langle y\rangle \frown \alpha))$	CSIscf4
$\lVert_{\mathrm{f}}^{k,l}(\langle x \trianglelefteq\ \mathsf{NT}(z)\ \trianglerighteq y\rangle \frown \alpha) =$ $(\lVert_{\mathrm{f}}^{k,1}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq\ \mathsf{NT}\ \trianglerighteq \lVert_{\mathrm{f}}^{k,1}(\alpha \frown \langle y\rangle)) \lhd k = l \rhd (\lVert_{\mathrm{f}}^{k,l+1}(\langle x\rangle \frown \alpha \frown \langle z\rangle) \trianglelefteq\ \mathsf{NT}\ \trianglerighteq \lVert_{\mathrm{f}}^{k,l+1}(\langle y\rangle \frown \alpha))$	CSIscf5

In CSIscf4, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$ different from the actions of the form $\mathsf{NT}(x)$ and the action NT.

3.7. Blocked thread forking

One can imagine that thread forking is only temporarily blocked if it is disabled. This motivates an interleaving strategy that postpones, when thread forking is disabled, the processing of a fork action $\mathsf{NT}(x)$ until thread forking is again enabled. This implies that the thread containing the fork action can proceed in only one way, for the processing of the fork action never fails. For this strategy, axiom CSIf5 from Table 15 must be replaced by axiom CSIbf5 from Table 17. Here, an additional test action $?\mathsf{NT}$ is used. It produces a positive reply if thread forking is enabled and it produces a negative reply if thread forking is disabled. The enabledness condition may, for example, be that the number of active threads is less than a maximum imposed on it. The modified strategic interleaving operator has not been given a special name because another strategy will be developed below, which is proposed as the more canonical approach.

3.8. Separating blocked thread forking from failed thread forking

The preceding strategy is only adequate if enabledness of thread forking entails success of thread forking. Otherwise, a better strategy is one that separates blocked thread forking from failed thread forking. In such a strategy, thread forking may still fail if it is not blocked. For instance, thread forking may be considered blocked if a given maximum number of threads is already active, whereas it may be considered failed if there is not enough free memory space left at the time the thread forking must be carried out. Failure of thread forking is considered an exception. Blocking of thread forking by itself, even if all actice threads try to perform a fork action, does not lead to deadlock because thread forking may become enabled by external events at any time. Hence, the test action $?\mathsf{NT}$ is repeatedly performed until it succeeds.

In the case where we assume a multi-processor system, another interpretation of blocking and failure is consistent with these equations as well: thread forking is blocked if there is no free processor, whereas it fails if the number of active threads is too high (the latter being viewed as a design flaw).

Table 17. Axiom for cyclic interleaving with temporarily blocked forking

$\lVert_{\mathrm{f}}(\langle x \trianglelefteq\ \mathsf{NT}(z)\ \trianglerighteq y\rangle \frown \alpha) = \lVert_{\mathrm{f}}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq\ ?\mathsf{NT}\ \trianglerighteq \lVert_{\mathrm{f}}(\alpha \frown \langle x \trianglelefteq\ \mathsf{NT}(z)\ \trianglerighteq y\rangle)$	CSIbf5

Table 18. Axiom for cyclic interleaving with separation of blocked forking and failed forking

$$\|_{\mathrm{bff}}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) = (\|_{\mathrm{bff}}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|_{\mathrm{bff}}(\alpha \frown \langle y\rangle)) \trianglelefteq ?\mathsf{NT} \trianglerighteq \|_{\mathrm{bff}}(\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle) \quad \text{CSIbff5}$$

Table 19. Axiom for cyclic interleaving with step counting and separation of blocked forking and failed forking

$$\begin{aligned}&\|^{k,l}_{\mathrm{bff}}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) =\\ &\quad ((\|^{k,1}_{\mathrm{bff}}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|^{k,1}_{\mathrm{bff}}(\alpha \frown \langle y\rangle)) \lhd k = l \rhd (\|^{k,l+1}_{\mathrm{bff}}(\langle x\rangle \frown \alpha \frown \langle z\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|^{k,l+1}_{\mathrm{bff}}(\langle y\rangle \frown \alpha)))\\ &\quad \trianglelefteq ?\mathsf{NT} \trianglerighteq\\ &\quad \|^{k,1}_{\mathrm{bff}}(\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle) \qquad \text{CSIscbff5}\end{aligned}$$

For cyclic interleaving with separation of blocked forking and failed forking, the axioms are the axioms from Table 15 with the strategic interleaving operator $\|_{\mathrm{f}}(\_)$ renamed to $\|_{\mathrm{bff}}(\_)$ and axiom CSIf5 replaced by axiom CSIbff5 from Table 18. This is in our opinion the most convincing description of forking that can be found in the current setting. If thread forking is temporarily blocked, it is attempted again on the next turn of the thread trying to fork off a thread. Otherwise, thread forking may either succeed or fail.

Step counting can easily be added to the previous strategy. For cyclic interleaving with step counting and separation of blocked forking and failed forking, the axioms are the axioms from Table 16 with the strategic interleaving operators $\|^k_{\mathrm{f}}(\_)$ and $\|^{k,l}_{\mathrm{f}}(\_)$ renamed to $\|^k_{\mathrm{bff}}(\_)$ and $\|^{k,l}_{\mathrm{bff}}(\_)$, respectively, and axiom CSIscf5 replaced by axiom CSIscbff5 from Table 19. Notice that, if thread forking is temporarily blocked, step counting starts over again for the thread trying to fork off a thread. The alternative where step counting does not start over again is plausible as well.

3.9. Terminating a named thread

Threads may influence one another during their life-time. In this section, we treat a strategy based on cyclic interleaving that supports creation of named threads, termination of named threads, and testing whether a named thread is still alive. It will be assumed that (i) created threads are named by positive natural numbers, which should occur as the first parameter of the thread creation action $\mathsf{NT}(k, x)$, and (ii) forking is possible unless a thread with the intended name already exists. Threads present initially are all given name 0. If a thread with the name k already exists, then the action $\mathsf{NT}(k, x)$ has no effect. Any existing thread has the ability to terminate any existing created thread, which may be itself.

In a superscript of the strategic interleaving operator a vector of thread names is given, one for each thread in the thread vector in the corresponding ordering. Two modification operators on name vectors and thread vectors are needed to define the strategic interleaving operator. Firstly, $\beta - k$ is the sequence obtained from β by removing all occurrences of k if k occurs in β, and β itself otherwise. Secondly, $\rho_{\beta-k}(\alpha)$ is the sequence obtained from α by removing all threads named k if k occurs in β, and α itself otherwise. Because forking is possible only if a thread with the intended name does not already exist, there will be at most one occurrences of k in β and at most one thread named k in α. Equations for $\|^{\beta}_{\mathrm{fn}}(\_)$ are presented in Table 20 which is based on Table 15, ignoring the possibility that thread forking is blocking. In CSIfn4, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$ different from the actions of the form $\mathsf{NT}(k, x)$, the actions of the form $\mathsf{terminate}!k$, and the actions of the form $\mathsf{isalive}?k$.

3.10. Using programs rather than processes

Polarized processes, or threads in our setting, are semantic abstractions from programs. Using polarized processes, a significant independence from particular program notations is obtained. At the same time, it is useful to understand these matters also with some particular program notation at hand. For this purpose, the notation of the program algebra based programming language PGLC from [BL02] can be used. PGLC is close to existing assembly languages.

Table 20. Axioms for cyclic interleaving with forking and termination

$\Vert_{\mathrm{fn}}(\alpha) = \Vert^{\vec{0}}_{\mathrm{fn}}(\alpha)$	CSIfn0
$\Vert^{\beta}_{\mathrm{fn}}(\langle\,\rangle) = \mathsf{S}$	CSIfn1
$\Vert^{\langle s\rangle^\frown\beta}_{\mathrm{fn}}(\langle \mathsf{S}\rangle^\frown\alpha) = \Vert^{\beta}_{\mathrm{fn}}(\alpha)$	CSIfn2
$\Vert^{\langle s\rangle^\frown\beta}_{\mathrm{fn}}(\langle \mathsf{D}\rangle^\frown\alpha) = \mathsf{S}_{\mathsf{D}}(\Vert^{\beta}_{\mathrm{fn}}(\alpha))$	CSIfn3
$\Vert^{\langle s\rangle^\frown\beta}_{\mathrm{fn}}(\langle x \trianglelefteq\ a\ \trianglerighteq y\rangle^\frown\alpha) = \Vert^{\beta^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle x\rangle) \trianglelefteq\ a\ \trianglerighteq \Vert^{\beta^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle y\rangle)$	CSIfn4
$\Vert^{\langle s\rangle^\frown\beta}_{\mathrm{fn}}(\langle x \trianglelefteq\ \mathsf{NT}(k,z)\ \trianglerighteq y\rangle^\frown\alpha) = \mathsf{tau}\circ(\Vert^{\beta^\frown\langle k\rangle^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle z\rangle^\frown\langle x\rangle) \lhd k \notin \beta \rhd \Vert^{\beta^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle y\rangle))$	CSIfn5
$\Vert^{\langle s\rangle^\frown\beta}_{\mathrm{fn}}(\langle x \trianglelefteq\ \mathsf{terminate!}k\ \trianglerighteq y\rangle^\frown\alpha) = \mathsf{tau}\circ(\Vert^{\beta}_{\mathrm{fn}}(\alpha) \lhd k = s \rhd (\Vert^{\beta^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle y\rangle) \lhd k \notin \beta \rhd \Vert^{\beta-k^\frown\langle s\rangle}_{\mathrm{fn}}(\rho_{\beta-k}(\alpha)^\frown\langle x\rangle)))$	CSIfn6
$\Vert^{\langle s\rangle^\frown\beta}_{\mathrm{fn}}(\langle x \trianglelefteq\ \mathsf{isalive?}k\ \trianglerighteq y\rangle^\frown\alpha) = \mathsf{tau}\circ(\Vert^{\beta^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle x\rangle) \lhd (k = s \vee k \in \beta) \rhd \Vert^{\beta^\frown\langle s\rangle}_{\mathrm{fn}}(\alpha^\frown\langle y\rangle))$	CSIfn7

Table 21. Defining equations for behaviour extraction

$\mid i, u_1\,;\,\ldots\,;\,u_n \mid = \mathsf{S}$	if not $1 \leq i \leq n$
$\mid i, u_1\,;\,\ldots\,;\,u_n \mid = a\circ \mid i+1, u_1\,;\,\ldots\,;\,u_n \mid$	if $u_i = a$
$\mid i, u_1\,;\,\ldots\,;\,u_n \mid = \mid i+1, u_1\,;\,\ldots\,;\,u_n \mid \trianglelefteq\ a\ \trianglerighteq \mid i+2, u_1\,;\,\ldots\,;\,u_n \mid$	if $u_i = +a$
$\mid i, u_1\,;\,\ldots\,;\,u_n \mid = \mid i+2, u_1\,;\,\ldots\,;\,u_n \mid \trianglelefteq\ a\ \trianglerighteq \mid i+1, u_1\,;\,\ldots\,;\,u_n \mid$	if $u_i = -a$
$\mid i, u_1\,;\,\ldots\,;\,u_n \mid = \mid i+k, u_1\,;\,\ldots\,;\,u_n \mid$	if $u_i = \#k$
$\mid i, u_1\,;\,\ldots\,;\,u_n \mid = \mid max(0, i-k), u_1\,;\,\ldots\,;\,u_n \mid$	if $u_i = \backslash\#k$

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

- for each $a \in \mathfrak{I}$, a *positive test instruction* $+a$;
- for each $a \in \mathfrak{I}$, a *negative test instruction* $-a$;
- for each $a \in \mathfrak{I}$, a *void basic instruction* a;
- for each $k \in \mathbb{N}$, a *forward jump instruction* $\#k$;
- for each $k \in \mathbb{N}$, a *backward jump instruction* $\backslash\#k$;

PGLC programs have the form $u_1\,;\,\ldots\,;\,u_n$, where $u_1, \ldots, u_n$ are primitive instructions of PGLC.

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

The behaviour of a PGLC program is a polarized process. The function $\mid\_\mid_{\mathrm{pglc}}$ that maps each PGLC program to its behaviour is defined by $\mid u_1\,;\,\ldots\,;\,u_n\mid_{\mathrm{pglc}} = \mid 1, u_1\,;\,\ldots\,;\,u_n \mid$ where $\mid\_,\_\mid$ is defined by the equations given in Table 21. In this table, $u_1, \ldots, u_n$ are primitive instructions of PGLC, $a \in \mathfrak{I}$ and $k, i \in \mathbb{N}$. The equations given in Table 21 do not cover the case where there are cyclic chains of jump instructions. We stipulate that $\mid i, u_1\,;\,\ldots\,;\,u_n \mid = \mathsf{D}$ if u_i is a jump instruction contained in a cyclic chain of jump instructions. It is easy to see that the behaviour of each PGLC program is definable by a finite guarded system of recursion equations over BPPA. Moreover, each finite guarded system of recursion equations over BPPA can be translated to a PGLC program of which the behaviour is the solution of the finite guarded system of recursion equations concerned (cf. Sect. 5 of [BBP05]).

Now, we consider the case where fork instructions $\mathsf{fork}\,\#k$ (for $k \in \mathbb{N}$) are added to PGLC. The intuition is that the execution of a fork instruction $\mathsf{fork}\,\#k$ leads to the start-up of the execution of the program in which the fork instruction occurs from the k-th next instruction in parallel with the original execution of that program. The original execution proceeds as if a basic instruction is executed, but it may be affected by the execution started up in parallel. How it is actually affected depends upon the interleaving strategy used for parallel execution of programs.

The definition of the function $|\_|_{\mathrm{pglc}}$ needs the following additional equations in the case where fork instructions are added to PGLC:

$$|\, i, u_1 \,;\, \ldots \,;\, u_n \,| = \mathsf{NT}(|\, i+k, u_1 \,;\, \ldots \,;\, u_n \,|) \circ |\, i+1, u_1 \,;\, \ldots \,;\, u_n \,| \quad \text{if } u_i = \mathsf{fork}\,\#k\ ,$$

$$|\, i, u_1 \,;\, \ldots \,;\, u_n \,| = |\, i+1, u_1 \,;\, \ldots \,;\, u_n \,| \trianglelefteq \mathsf{NT}(|\, i+k, u_1 \,;\, \ldots \,;\, u_n \,|) \trianglerighteq |\, i+2, u_1 \,;\, \ldots \,;\, u_n \,| \quad \text{if } u_i = +\mathsf{fork}\,\#k\ ,$$

$$|\, i, u_1 \,;\, \ldots \,;\, u_n \,| = |\, i+2, u_1 \,;\, \ldots \,;\, u_n \,| \trianglelefteq \mathsf{NT}(|\, i+k, u_1 \,;\, \ldots \,;\, u_n \,|) \trianglerighteq |\, i+1, u_1 \,;\, \ldots \,;\, u_n \,| \quad \text{if } u_i = -\mathsf{fork}\,\#k\ .$$

In the presence of fork instructions, the real behaviour of a program $u_1 \,;\, \ldots \,;\, u_n$ is not $|\, u_1 \,;\, \ldots \,;\, u_n \,|_{\mathrm{pglc}}$. The real behaviour is $\|_{si}(|\, u_1 \,;\, \ldots \,;\, u_n \,|_{\mathrm{pglc}})$, where $\|_{si}(\_)$ is the strategic interleaving operator corresponding to the interleaving strategy used for parallel execution of programs.

4. Interleaving strategies for blocking actions

In general, a basic action is a command to be processed by a specific service offered by the execution architecture and that service may be in a state where not all commands are enabled. This leads to blocking actions, i.e. actions that block threads. For example, a para-target shared service with locking (introduced in Sect. 5) has either its locking action or its unlocking action enabled, but not both. Thus, as intended, successive lockings of the service without intermediate unlockings leads to blocked threads. In this section, we treat three strategies that can deal with blocking actions. Preceding, we introduce actions with 'foci' to indicate the service for which they represent commands and special actions for testing the enabledness of commands.

4.1. Foci, methods and guards

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*. For the set $\mathcal{A}$ of basic actions, we take the set

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

Each focus plays the role of a name of a service provided by the execution architecture that can be requested to process a command. Each method plays the role of a command proper. Performing a basic action $f.m$ is taken as making a request to the service with focus f to process the command m. Performing a basic action $f?m$ is taken as making the request to the service with focus f to reply whether it is able to process method m. Performing a basic action $f?m$ can be regarded as performing an enabledness test for $f.m$. If it produces a positive reply and $f.m$ is performed next, $f.m$ will not block the thread concerned. If it produces a negative reply and $f.m$ is performed next, $f.m$ will block the thread concerned. If it produces a positive reply, m is said to be enabled. Basic action of the form $f?m$ are shortly called *guards*.

It is assumed that processing $f.m$ does not change the state of any service other than the one with focus f and processing $f?m$ does not change the state of any service. The action tau, which arises as the residue of processing enabled actions, does not change the state of any service either. Moreover, tau is not connected to a specific focus and is always enabled.

Threads are supposed initially not to contain any guards. It is the task of the interleaving strategy to deal with blocking actions by making use of guards. In other words, the strategic interleaving operators may introduce guards.

Recall that services are classified into three categories: para-target local services, para-target shared services and target shared services. It is assumed that this classification can be derived from the foci, so that three sets of foci can be distinguished: (i) $\mathcal{F}_{\mathrm{ptls}}$ is the set of all foci under which a para-target local service is known, (ii) $\mathcal{F}_{\mathrm{ptss}}$ is the set of all foci under which a para-target shared service is known, and (iii) $\mathcal{F}_{\mathrm{tss}}$ is the set of all foci under which a target shared service is known. For a focus in $\mathcal{F}_{\mathrm{ptls}}$, it is assumed that there exists at most one thread that contains actions in which that focus occurs.

Table 22. Axioms for cyclic interleaving with blocking actions

$\lVert_{\mathrm{ba}}(\langle\,\rangle) = \mathsf{S}$	CSIba1
$\lVert_{\mathrm{ba}}(\langle \mathsf{S}\rangle \frown \alpha) = \lVert_{\mathrm{ba}}(\alpha)$	CSIba2
$\lVert_{\mathrm{ba}}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\lVert_{\mathrm{ba}}(\alpha))$	CSIba3
$\lVert_{\mathrm{ba}}(\langle \mathsf{tau} \circ x\rangle \frown \alpha) = \mathsf{tau} \circ \lVert_{\mathrm{ba}}(\alpha \frown \langle x\rangle)$	CSIba4
$\lVert_{\mathrm{ba}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = (\lVert_{\mathrm{ba}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \lVert_{\mathrm{ba}}(\alpha \frown \langle y\rangle)) \trianglelefteq f?m \trianglerighteq \lVert_{\mathrm{ba}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	CSIba5

4.2. Dealing with blocking actions

To begin with, we introduce the simplest cyclic interleaving strategy that can deal with blocking actions. It is a strategy without step counting, and forking is not supported. Moreover, it has some shortcomings which will be discussed below. The axioms for simple cyclic interleaving with blocking actions ($\lVert_{\mathrm{ba}}(\_)$) are given in Table 22.

This interleaving strategy has two obvious shortcomings: (i) if all threads are blocked, this is not detected and (ii) the same enabledness test may be performed twice or more without any state change in between. These shortcomings can be solved by keeping a count of the number of threads that are certainly blocked and maintaining the set of all actions of the form $f.m$, with $f \in \mathcal{F}_{\mathrm{ptss}}$, for which the corresponding enabledness test has produced a negative reply after the processing of the last action of the form $f.m'$. As explained below, the enabledness test for $f.m$ need not be repeated before the processing of the next action of the form $f.m'$ if $f.m$ is in this set.

4.3. Avoiding redundant tests with memory

For a focus f in $\mathcal{F}_{\mathrm{ptls}}$, if an enabledness test for $f.m$ has produced a negative reply, then the thread on which behalf the enabledness test has been performed should deadlock. The alternative is to proceed with performing it again. However, performing $f?m$ again would produce the same reply because neither another thread nor an external factor can influence the state of the para-target local service with focus f.

For a focus f in $\mathcal{F}_{\mathrm{ptss}}$, if an enabledness test for $f.m$ has produced a negative reply, the state of the service with focus f may change only when that service processes an action of the form $f.m'$ for another thread. For that reason, it is practical to perform the enabledness test for $f.m$ again only after an action of the form $f.m'$ has been processed by the service with focus f. For that purpose, the actions of the form $f.m$ for which the corresponding enabledness test has produced a negative reply after the processing of the last action of the form $f.m'$ should be recorded. If $f.m$ is recorded, the enabledness tests for it need not be repeated before the processing of the next action of the form $f.m'$. When an action of the form $f.m'$ is performed, all recorded actions of the form $f.m$ should be dropped as the corresponding enabledness tests may now produce a positive reply.

For a focus f in $\mathcal{F}_{\mathrm{tss}}$, if an enabledness test for $f.m$ has produced a negative reply, the state of the service with focus f may change at any stage by external causes. For that reason, it is practical to repeat the enabledness test for $f.m$ at the next turn of the thread concerned, thus waiting for a moment at which the action $f.m$ is enabled. Hence, there is no reason to record any outcome of enabledness tests.

The axioms for a cyclic interleaving strategy which avoids redundant test ($\lVert^{V,\beta}_{\mathrm{bam}}(\_)$) as described above are given in Table 23. Because this interleaving strategy has to memorize the outcome of enabledness tests to a certain extent, we call it an interleaving strategy with memory. The notation $f.$ is used for the set $\{f.m \mid m \in \mathcal{M}\}$. The superscript V is a finite set of actions of the form $f.m$, which is used to memorize the outcome of enabledness tests as described above. The superscript β is a vector of bits, one for each thread in the thread vector in the corresponding ordering. This bit vector keeps the threads which have been found blocked and for which this judgment is still valid with certainty. Only if a thread is found blocked at a request for a para-target shared service is the corresponding bit in this vector set to 1. If a target service is found blocked, busy waiting takes place because the state of the target service may change at any stage by some external cause such that the blocking comes to an end – so even without any intermediate activity of threads in the thread vector. In CSIbam6, all bits in the bit vector are reset to 0 after an action has been processed by a para-target shared service because it is not known which threads have been blocked by actions for the same service and the worst case must be taken into account. If all threads are certainly blocked, i.e. all bits in the bit vector are set to 1, deadlock is unavoidable.

Step counting can easily be added to the previous strategy. The axioms for this strategy are given in Table 24. Thread forking has been added as well. The underlying conceptual decision is that deadlock can occur only if

Table 23. Axioms for cyclic interleaving with blocking actions and memory

$$\|_{\mathrm{bam}}(\alpha) = \|^{\emptyset,\vec{0}}_{\mathrm{bam}}(\alpha) \qquad \text{CSIbam0}$$

$$\|^{V,\beta}_{\mathrm{bam}}(\langle\rangle) = \mathsf{S} \qquad \text{CSIbam1}$$

$$\|^{V,\langle b\rangle\frown\beta}_{\mathrm{bam}}(\langle \mathsf{S}\rangle \frown \alpha) = \|^{V,\beta}_{\mathrm{bam}}(\alpha) \qquad \text{CSIbam2}$$

$$\|^{V,\langle b\rangle\frown\beta}_{\mathrm{bam}}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\|^{V,\beta}_{\mathrm{bam}}(\alpha)) \qquad \text{CSIbam3}$$

$$\|^{V,\langle b\rangle\frown\beta}_{\mathrm{bam}}(\langle \mathsf{tau}\circ x\rangle \frown \alpha) = \mathsf{tau}\circ \|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle x\rangle) \qquad \text{CSIbam4}$$

$$\|^{V,\langle b\rangle\frown\beta}_{\mathrm{bam}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = (\|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle y\rangle)) \trianglelefteq f?m \trianglerighteq \|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle \mathsf{D}\rangle) \quad \text{if } f \in \mathcal{F}_{\mathrm{ptls}} \qquad \text{CSIbam5}$$

$$\|^{V,\langle b\rangle\frown\beta}_{\mathrm{bam}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = \mathsf{D} \trianglelefteq \langle b\rangle \frown \beta = \vec{1} \trianglerighteq (\|^{V,\beta\frown\langle 1\rangle}_{\mathrm{bam}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle) \trianglelefteq f.m \in V \trianglerighteq ((\|^{V-f.,\vec{0}}_{\mathrm{bam}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V-f.,\vec{0}}_{\mathrm{bam}}(\alpha \frown \langle y\rangle)) \trianglelefteq f?m \trianglerighteq \|^{V\cup\{f.m\},\beta\frown\langle 1\rangle}_{\mathrm{bam}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle))) \quad \text{if } f \in \mathcal{F}_{\mathrm{ptss}} \qquad \text{CSIbam6}$$

$$\|^{V,\langle b\rangle\frown\beta}_{\mathrm{bam}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = (\|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle y\rangle)) \trianglelefteq f?m \trianglerighteq \|^{V,\beta\frown\langle b\rangle}_{\mathrm{bam}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle) \quad \text{if } f \in \mathcal{F}_{\mathrm{tss}} \qquad \text{CSIbam7}$$

all threads are blocked and none of the threads attempts to fork off a thread. Notice that $\|^{1}_{\mathrm{bafm}}(\_)$ coincides with $\|_{\mathrm{bam}}(\_)$ on all thread vectors that do not involve thread forking.

5. Locking, unlocking and deadlock

Interesting deadlocks arise from the coexistence of a number of threads that interact with shared services that may block threads. A para-target shared service with locking is an example of such a service. In this section, we consider some thread vectors of which the threads interact with a para-target shared service with locking, and show that it may depend on the ordering of the threads in the thread vector or the interleaving strategy whether the interleaving of the thread vector leads to deadlock. The reasoning in this section is entirely informal. Formalized proofs require some additional formalization. This is postponed till Sect. 7.

A para-target service does nothing but maintain a state and produce replies on the basis of that state. A para-target shared service with locking is equipped with two methods lock and unlock. In every state, either of the two methods is enabled and processing of the enabled one moves the service to a state in which the other one is enabled. In this way, it prevents successive lockings without intermediate unlocking. It is assumed that initially lock is enabled.

All threads are supposed to work as follows. If a thread successfully performs the action f.lock, it acquires the lock on the service with focus f. It keeps the lock until it is released by performing f.unlock. For each focus f in $\mathcal{F}_{\mathrm{ptss}}$, all commands $f.m$ must be performed in a phase in which the thread keeps the lock of the service with focus f. If all threads adhere to this rule (a matter to be ensured by the procedures for generating and/or accepting threads by the system) and all para-target shared services are para-target shared service with locking, it is guaranteed that (i) at most a single thread can keep the lock on a para-target shared service at a given stage and (ii) a thread keeping the lock on a para-target shared service has exclusive access to that service. Henceforth, we will say 'the lock on f' if we mean 'the lock on the service with focus f'.

5.1. Single thread deadlock

By compromising the requirement that threads perform locking and unlocking in an alternating order, a single thread may deadlock. Consider

$$P = \|_{\mathrm{bam}}(\langle f.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle)\ .$$

Table 24. Axioms for cyclic interleaving with step counting, forking, blocking actions, and memory

$$\|^{k}_{\mathrm{bafm}}(x) = \|^{\emptyset,k,1,\vec{0}}_{\mathrm{bafm}}(x) \qquad \text{CSIscbafm0}$$

$$\|^{V,k,l,\beta}_{\mathrm{bafm}}(\langle\rangle) = \mathsf{S} \qquad \text{CSIscbafm1}$$

$$\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle \mathsf{S}\rangle \frown \alpha) = \|^{V,k,1,\beta}_{\mathrm{bafm}}(\alpha) \qquad \text{CSIscbafm2}$$

$$\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\|^{V,k,1,\beta}_{\mathrm{bafm}}(\alpha)) \qquad \text{CSIscbafm3}$$

$$\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle \mathsf{tau}\circ x\rangle \frown \alpha) = \mathsf{tau}\circ(\|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x\rangle) \lhd k = l \rhd \|^{V,k,l+1,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x\rangle \frown \alpha)) \qquad \text{CSIscbafm4}$$

$$\begin{aligned}
&\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = \\
&\quad((\|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle y\rangle)) \\
&\qquad \lhd k = l \rhd \\
&\quad(\|^{V,k,l+1,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x\rangle \frown \alpha) \trianglelefteq f.m \trianglerighteq \|^{V,k,l+1,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle y\rangle \frown \alpha))) \\
&\quad\trianglelefteq f?m \trianglerighteq \\
&\quad\|^{V,k,l,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle \mathsf{D}\rangle) \qquad \text{if } f \in \mathcal{F}_{\mathrm{ptls}}
\end{aligned} \qquad \text{CSIscbafm5}$$

$$\begin{aligned}
&\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = \\
&\quad\mathsf{D} \\
&\quad\lhd \langle b\rangle \frown \beta = \vec{1} \rhd \\
&\quad(\|^{V,k,1,\beta\frown\langle 1\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle) \\
&\qquad\lhd f.m \in V \rhd \\
&\quad((((\|^{V-f.,k,1,\vec{0}}_{\mathrm{bafm}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V-f.,k,1,\vec{0}}_{\mathrm{bafm}}(\alpha \frown \langle y\rangle))) \\
&\qquad\lhd k = l \rhd \\
&\quad(\|^{V-f.,k,l+1,\vec{0}}_{\mathrm{bafm}}(\langle x\rangle \frown \alpha) \trianglelefteq f.m \trianglerighteq \|^{V-f.,k,l+1,\vec{0}}_{\mathrm{bafm}}(\langle y\rangle \frown \alpha))) \\
&\quad\trianglelefteq f?m \trianglerighteq \\
&\quad\|^{V\cup\{f.m\},k,l,\beta\frown\langle 1\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle))))) \qquad \text{if } f \in \mathcal{F}_{\mathrm{ptss}}
\end{aligned} \qquad \text{CSIscbafm6}$$

$$\begin{aligned}
&\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = \\
&\quad((\|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle y\rangle)) \\
&\qquad\lhd k = l \rhd \\
&\quad(\|^{V,k,l+1,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x\rangle \frown \alpha) \trianglelefteq f.m \trianglerighteq \|^{V,k,l+1,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle y\rangle \frown \alpha))) \\
&\quad\trianglelefteq f?m \trianglerighteq \\
&\quad(\|^{V,k,l,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)) \qquad \text{if } f \in \mathcal{F}_{\mathrm{tss}}
\end{aligned} \qquad \text{CSIscbafm7}$$

$$\begin{aligned}
&\|^{V,k,l,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) = \\
&\quad((\|^{V,k,1,\beta\frown\langle 0\rangle\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle y\rangle)) \\
&\qquad\lhd k = l \rhd \\
&\quad(\|^{V,k,l+1,\langle b\rangle\frown\beta\frown\langle 0\rangle}_{\mathrm{bafm}}(\langle x\rangle \frown \alpha \frown \langle z\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|^{V,k,l+1,\langle b\rangle\frown\beta}_{\mathrm{bafm}}(\langle y\rangle \frown \alpha))) \\
&\quad\trianglelefteq ?\mathsf{NT} \trianglerighteq \\
&\quad\|^{V,k,1,\beta\frown\langle b\rangle}_{\mathrm{bafm}}(\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)
\end{aligned} \qquad \text{CSIscbafm8}$$

In P, the second attempt to acquire the lock on f fails and the thread will deadlock. By requiring that threads alternate locking and unlocking on the same service, single thread deadlock disappears as in the case of Java which features no single thread deadlock either.

5.2. Deadlock behavior and its dependence on thread-order

We say that a thread vector deadlocks given some strategy if that strategy acting on it produces a process ending in D. This may depend on the replies from target shared services. For instance, if $f \in \mathcal{F}_{\mathrm{tss}}$, the thread vector $\langle \mathsf{S} \trianglelefteq f.m \trianglerighteq \mathsf{D}\rangle$ will deadlock under $\|_{\mathrm{bam}}(\_)$ after a negative reply by the service with focus f. After a positive reply it terminates successfully, and in the case of blocking it waits till $f.m$ is found enabled.

More interesting deadlocks arise from the coexistence of a number of threads that interact with shared services that may block threads. Avoiding such deadlocks is entirely in the hands of the designer of the threads in a thread vector, i.e. the programmer of a multi-threaded program. An archetypical example is the following:

$$P = \|_{\mathrm{bam}}(\langle f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle \frown \langle g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle) \ ,$$

where f and g are foci of different para-target shared services with locking. In this example, deadlock occurs because the cyclic interleaving strategy allows both threads to perform their initial locking actions and subsequently neither thread is able to proceed with the next locking action.

Let h be the focus of a para-target shared service with a method idle which is never blocked and will not cause any change of state. So idle is like tau but it needs a focus, unlike tau. Now the following system deadlocks as well:

$$Q = \|_{\mathrm{bam}}(\langle h.\mathsf{idle} \circ f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle \frown \langle g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle) \ .$$

In this example, deadlock occurs for the same reason as in the previous example, taking into account that the first thread makes one redundant step.

By placing both threads in a different order in the thread vector, however, deadlock does not occur, because now the first thread acquires both locks before the second thread is able to perform a locking action:

$$R = \|_{\mathrm{bam}}(\langle g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle \frown \langle h.\mathsf{idle} \circ f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle) \ .$$

Indeed, in the case of R, the step $h.\mathsf{idle}$ prevents the second thread from prematurely acquiring the lock that the first thread needs to proceed.

5.3. Deadlock behavior and its dependence on interleaving strategy

Deadlock behavior is dependent on the interleaving strategy as well as on thread ordering. To substantiate this point, we will use the step counting version of the interleaving strategy with memory to avoid redundant tests in dealing with blocking actions (see Sect. 4.3).

In connection with deadlock behavior the following can be observed:

1. there is a thread vector α such that $\|^1_{\mathrm{bafm}}(\alpha)$ runs into deadlock whereas $\|^2_{\mathrm{bafm}}(\alpha)$ does not;
2. there is a thread vector β such that $\|^2_{\mathrm{bafm}}(\beta)$ runs into deadlock whereas $\|^1_{\mathrm{bafm}}(\beta)$ does not.

For α one may simply take the thread vector used in P in Sect. 5.2. The following example for β combines ideas from the examples in Sect. 5.2:

$$\begin{aligned}
\beta &= X \frown Y \ , \\
X &= \langle e.\mathsf{lock} \circ e.\mathsf{unlock} \circ f.\mathsf{lock} \circ h.\mathsf{idle} \circ g.\mathsf{lock} \circ h.\mathsf{idle} \circ g.\mathsf{unlock} \circ h.\mathsf{idle} \circ f.\mathsf{unlock} \circ h.\mathsf{idle} \circ \mathsf{S}\rangle \ , \\
Y &= \langle e.\mathsf{lock} \circ e.\mathsf{unlock} \circ h.\mathsf{idle} \circ g.\mathsf{lock} \circ f.\mathsf{lock} \circ h.\mathsf{idle} \circ f.\mathsf{unlock} \circ h.\mathsf{idle} \circ g.\mathsf{unlock} \circ h.\mathsf{idle} \circ \mathsf{S}\rangle \ .
\end{aligned}$$

In $\|^2_{\mathrm{bafm}}(\beta)$, in the first round, first thread X acquires and subsequently releases the lock on e and next thread Y acquires and subsequently releases the lock on e. In the second round, thread X acquires the lock on f and thread Y acquires the lock on g. In the third round, both threads are blocked. In $\|^1_{\mathrm{bafm}}(\beta)$, in the first round, thread X acquires the lock on e and thread Y is blocked. In the second round, thread X releases the lock on e and thread Y acquires the lock on e. In the third round, thread X acquires the lock on f and thread Y releases the lock on e. Then both threads make a redundant step and after that thread X acquires the lock on g and thread Y is blocked again. Because of the particular setup, the one step strategy causes thread Y to be processed so much slower then the two step strategy, in comparison to thread X, that thread X can keep the locks on both f and g at the same time with the one step strategy, but not with the two step strategy.

5.4. Examples without an explicit thread vector

One might say that the examples from Sects. 5.1–5.3 make use of an element foreign to common programming, namely the explicit construction of a thread vector. Therefore, we adapt some examples to polarized processes

with forking. Forking gives rise to thread vectors, but no 'user control' of thread vectors is presupposed. It is taken for granted that forking always succeeds. We adapt P and R from Sect. 5.2 as follows:

$$P' = \|^1_{\mathrm{bafm}}(g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S} \trianglelefteq \mathsf{NT}(f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}) \trianglerighteq \mathsf{S}) ,$$
$$R' = \|^1_{\mathrm{bafm}}(h.\mathsf{idle} \circ f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S} \trianglelefteq \mathsf{NT}(g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}) \trianglerighteq \mathsf{S}) .$$

The process P' generates after one step of a cyclic non-counting strategy the initial configuration of P from Sect. 5.2, and it will deadlock just as P. The process R' generates after one step of a cyclic non-counting strategy the initial configuration of R from Sect. 5.2, and it will not deadlock just as R.

5.5. A methodological implication

A consequence of the observations made in Sect. 5.3 is that a manifestation of deadlock for some thread vector on a system providing one interleaving strategy, has no implications for the deadlock behavior of the same thread vector on a system providing another interleaving strategy. This makes it rather difficult to provide a definition of deadlock that is independent of the interleaving strategy. The only reasonable option is to say that a thread vector leads to deadlock in some context if for some (adequate) interleaving strategy it ends up in deadlock.

This analysis of deadlock, however, requires one to specify the class of interleaving strategies, and then to evaluate an existential quantifier over that class. Clearly, interleaving strategies must be in some sense correct. Our claim is then this: for the programmer working with multi-threading, it is a plausible assumption that he/she has in mind a number of examples of correct interleaving strategies, in the way some of these have been specified above. However, the assumption is unwarranted that he/she has in mind a general survey theory of strategic interleaving which permits a precise interpretation of the mentioned existential quantifier needed.

As a consequence, the notion of deadlock as a phenomenon visible or comprehensible at an abstraction level at which no definite specification of the interleaving strategy has yet been given may be considered problematic. For that reason, the concept of deadlock is best understood as an operational phenomenon at an abstraction level of strategic interleaving, i.e. given one or a number of correct interleaving strategies for the program notation at hand. Stated differently: deadlock behavior is a strategy dependent concept from the theory of strategic interleaving for multi-threaded systems.

Having available a significant collection of strategies, one may intend to enforce deadlock freedom with respect of these strategies. Step counting strategies involve a numerical parameter, and in general more of such parameters may occur, thus leading to a combinatorial explosion of strategies of a certain form. Using automated tests, a large set of interleaving strategies can be dealt with, thus producing empirical evidence of deadlock freedom at least. To predict deadlock freedom with complete certainty, however, a complete grasp of the possible strategies which may be used at run-time is necessary.

6. Execution architectures for multi-threading

In this short section, we recapitulate and amplify some earlier remarks, concerning our understanding of execution architectures for multi-threading, that are helpful in appreciating the definitions given in Sect. 7.

Essentially, we adopt the setup of execution architectures proposed in [BP07] and extend it from the single thread case to the case involving a thread vector. Thus, an execution architecture is given by (i) a thread vector together with (ii) a strategic interleaving operator, and (iii) a number of services known within the architecture under their focus. After application of a strategic interleaving operator, a polarized process is obtained which operates in the context of the services as proposed in [BP07].

Suppose that the execution architecture contains a service F known under focus f. Then, whenever the multi-thread issues a command $f.m$, control is given to F to process method m. Processing m may lead to a state change in F. It may also involve interaction of F with the environment of the execution architecture. After the processing of m is completed, control is returned to the multi-thread issuing the command $f.m$, together with a mandatory reply value.

A service has no interaction with other services in the execution architecture. A para-target service has no interaction with the environment of the execution architecture either, and it is deterministic in its replies to the multi-thread issuing commands. A typical example of a para-target service is a service that keeps shared data

for a number of threads in the multi-thread. Para-target services are 'used' by the multi-thread in the sense that commands for a para-target service are only issued in order to make use of the reply that is returned.

7. Formalizing the phenomenon of deadlock

Para-target services and their use by multi-threads were essential in the examples from Sects. 5.2 and 5.3 concerning the dependence of deadlock behavior on thread-order and interleaving strategy. Nevertheless, para-target services and their use by multi-threads were kept entirely informal in those examples. In this section, we make precise what it means that a multi-thread leads to deadlock in some context of para-target services. Preceding that, we introduce a representation of para-target services and an operator to describe the behavior of multi-threads that use para-target services. We also use that operator to specify an interleaving strategy where, at thread creation, a thread gets a certain para-target local service, with a certain initial state.

7.1. Represention of para-target services

Para-target services extend the state machine behaviors of [BP02] with the possibility that there are states in which not all commands are enabled. Because a para-target service has a pure auxiliary role and interacts only with threads, its behavior may be formalized, following [BP02], by means of a reply function. The definition of reply functions below takes enabledness into account.

A para-target service is represented by a function $F : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$ with the property that $F(\alpha) = \mathsf{B} \Rightarrow F(\alpha \frown \langle m \rangle) = \mathsf{B}$ for all $\alpha \in \mathcal{M}^+$ and $m \in \mathcal{M}$. This function is called the *reply* function of the para-target service.

Given a reply function F and a method m, the derived reply function of F after processing m, written $\frac{\partial}{\partial m}F$, is defined by $\frac{\partial}{\partial m}F(\alpha) = F(\langle m \rangle \frown \alpha)$.

The connection between a reply function F and the para-target service represented by it can be understood as follows:

- if $F(\langle m \rangle) = \mathsf{T}$, m is accepted by the service, the reply is positive, and the service proceeds as $\frac{\partial}{\partial m}F$;
- if $F(\langle m \rangle) = \mathsf{F}$, m is accepted by the service, the reply is negative, and the service proceeds as $\frac{\partial}{\partial m}F$;
- If $F(\langle m \rangle) = \mathsf{B}$, m is not accepted by the service.

If m is accepted by the service, m is called enabled. If m is not accepted by the service, the thread issuing m is called blocked. Henceforth, we will identify a reply function with the para-target service represented by it.

7.2. Use operators for para-target services

For each $f \in \mathcal{F}_{\mathrm{ptls}} \cup \mathcal{F}_{\mathrm{ptss}}$, we introduce a *use* operator $\_ /_f \_$. These operators have a polarized process as first argument and a para-target service as second argument. $x /_f F$ is the polarized process that results if all basic actions of the form $f.m$ or $f?m$ performed by polarized process x are taken as commands to be processed by service F. In the case where m is not accepted, $f.m$ leads to deadlock and $f?m$ yields a negative reply. The action tau arises as the residue of processing commands. Every service accepts tau. The defining equations for the use operators are in Table 25. In Table 26, an additional axiom for the conditional operator is given because use is made of the additional truth value B.

The use operators permit encapsulation of the para-target services in the polarized process representing the thread vector after strategic interleaving. This is justified because the para-target services are used only by the threads in the thread vector. Two use operator applications are independent if they concern different foci. In that case the order of application does not matter.

Theorem (Commuting Uses.) $f \neq g \Rightarrow (x /_f F) /_g G = (x /_g G) /_f F$.

Proof. This has been demonstrated in a setting without blocking methods in [BP02]. In spite of the fact that several definitions have a somewhat different form in that paper, the proof carries over without difficulty and will not be redone here. □

In the case of dependence, one of the two services is redundant: $(x /_f F) /_f G = x /_f F$.

Table 25. Axioms for use operators

$\mathsf{S} \mathbin{/_f} H = \mathsf{S}$	use1
$\mathsf{D} \mathbin{/_f} H = \mathsf{D}$	use2
$(\mathsf{tau} \circ x) \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} H)$	use3
$f \neq g \Rightarrow (x \trianglelefteq g.m \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq g.m \trianglerighteq (y \mathbin{/_f} H)$	use4
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ ((x \lhd H(\langle m \rangle) \rhd y) \mathbin{/_f} \frac{\partial}{\partial m} H)$	use5
$f \neq g \Rightarrow (x \trianglelefteq g?m \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq g?m \trianglerighteq (y \mathbin{/_f} H)$	use6
$(x \trianglelefteq f?m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ ((x \lhd H(\langle m \rangle) = \mathsf{T} \vee H(\langle m \rangle) = \mathsf{F} \rhd y) \mathbin{/_f} H)$	use7

Table 26. Additional axiom for conditional operator

$x \lhd \mathsf{B} \rhd y = \mathsf{D}$	CO3

7.3. The phenomenon of deadlock

The use operators permit us to give a precise definition of deadlock behavior. This requires a notation for repeated internal steps. For each term t of thread algebra and each $k \geq 0$, the term $\mathsf{tau}^k(t)$ is defined by induction on k as follows:

$$\mathsf{tau}^0(t) = t\,,$$
$$\mathsf{tau}^{k+1}(t) = \mathsf{tau} \circ \mathsf{tau}^k(t)\,.$$

A multi-thread P in the context of para-target services $F_1,\ldots,F_n$, under foci $f_1,\ldots,f_n$, shows deadlock if for some k, $P_{\mathrm{context}} = ((P \mathbin{/_{f_1}} F_1) \mathbin{/_{f_2}} F_2) \ldots \mathbin{/_{f_n}} F_n = \mathsf{tau}^k(\mathsf{D})$.[4]

Besides deadlock, there may be livelock or proper termination (also called convergence). Livelock occurs if $P_{\mathrm{context}} = \mathsf{tau} \circ P_{\mathrm{context}}$, and proper termination takes place if $P_{\mathrm{context}} = \mathsf{tau}^k(\mathsf{S})$ or $P_{\mathrm{context}} = \mathsf{tau}^k(g.m \circ Q)$ for some basic action $g.m$ with $g \notin \{f_1, \ldots, f_n\}$.

Proposition (Correctness of Deadlock Examples) This formalization and the axioms given for the operators concerned suffice to prove all claims made about the presence and absence of deadlock in Sects. 5.1, 5.2 and 5.3.

Proof. The proof of each claim amounts to a careful equational rewriting. We show the proof of only the first claim made in Sect. 5.2:

$$\begin{aligned}
&(\|_{\mathrm{bam}}^{\emptyset,\langle 0\rangle^\frown\langle 0\rangle}(\langle f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle \curvearrowright \\
&\qquad \langle g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle) \mathbin{/_f} F) \mathbin{/_g} G \\
&= \mathsf{tau}^2((\|_{\mathrm{bam}}^{\emptyset,\langle 0\rangle^\frown\langle 0\rangle}(\langle g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle \curvearrowright \\
&\qquad \langle g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle) \mathbin{/_f} F) \mathbin{/_g} G) \\
&= \mathsf{tau}^4((\|_{\mathrm{bam}}^{\emptyset,\langle 0\rangle^\frown\langle 0\rangle}(\langle g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle \curvearrowright \\
&\qquad \langle f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle) \mathbin{/_f} F) \mathbin{/_g} G) \\
&= \mathsf{tau}^5((\|_{\mathrm{bam}}^{\{g.\mathsf{lock}\},\langle 0\rangle^\frown\langle 1\rangle}(\langle f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle \curvearrowright \\
&\qquad \langle g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle) \mathbin{/_f} F) \mathbin{/_g} G) \\
&= \mathsf{tau}^6((\|_{\mathrm{bam}}^{\{f.\mathsf{lock},g.\mathsf{lock}\},\langle 1\rangle^\frown\langle 1\rangle}(\langle g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle \curvearrowright \\
&\qquad \langle f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle) \mathbin{/_f} F) \mathbin{/_g} G) \\
&= \mathsf{tau}^6(\mathsf{D})\,.
\end{aligned}$$

In each step, axiom CSIbam6 is applied. In addition, axioms use3–use7 are applied in the first two steps, axioms use3 and use6–use7 are applied in the second two steps, and axiom use2 is applied in the last step. Moreover, use

[4] As D is not an action in most program notations it is taken for granted that the semantic translation from a program to its thread will not by itself produce an occurrence of D. In the setting of thread algebra interesting examples of deadlock should involve thread vectors with D-free threads.

Table 27. Additional axiom for cyclic interleaving with blocked forking, failed forking and initialization of a local service

$\lVert_{\mathrm{bff}}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) = (\lVert_{\mathrm{bff}}(\alpha \frown \langle z /_f F_{init}\rangle \frown \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \lVert_{\mathrm{bff}}(\alpha \frown \langle y\rangle)) \trianglelefteq ?\mathsf{NT} \trianglerighteq \lVert_{\mathrm{bff}}(\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)$	CSIbffi5

is made of the following properties of the locking mechanism of para-target services with locking:

$$H(\alpha \frown \langle \mathsf{lock}\rangle) = \mathsf{T} \Leftrightarrow \#_{\mathsf{lock}}(\alpha) - \#_{\mathsf{unlock}}(\alpha) = 0 \,,$$
$$H(\alpha \frown \langle \mathsf{lock}\rangle) = \mathsf{B} \Leftrightarrow \#_{\mathsf{lock}}(\alpha) - \#_{\mathsf{unlock}}(\alpha) \neq 0 \,,$$

for all reply functions H and sequences of methods α. Here, $\#_m(\alpha)$ denotes the number of occurrences of m in α. The proofs of the other claims made in Sects. 5.2 and 5.3 are similar, but in the case of the claims made in Sect. 5.3 the proofs are rather long. □

7.4. Initialized para-target local services

Suppose that each thread is granted the use of a para-target local service F, with focus f, with initial state F_{init}. In this case, a strategy can be specified with the help of a use operator. If the matter is considered without step counting and blocking actions, it amounts to the replacement of the equation for thread creation from Table 18 by the equation given in Table 27.[5]

A weakness of the setup in Table 27 is that F_{init} lies outside the algebraic framework of the thread algebra. It would be more systematic if an algebra of services is used to denote various services.

For the purpose of the description of multi-threaded systems, the technique used to specify services is immaterial, however. For that reason, no further exposition of the service algebra will be given here.

8. Classes and class methods

The term method in its connotation in object-oriented programming should be distinguished from its connotation in the preceding sections. In this section, it will be used in the former connotation.

Although synchronization of methods is just a matter of locking and unlocking, several particular phenomena take place. If a thread has acquired a lock, any subsequent attempt made by the thread to acquire the same lock again without releasing it in between succeeds. This is needed to deal with recursion in the form of recursive method calls, which plays an essential role in object-oriented programming languages. In contrast, in the example from Sect. 5.1, deadlock occurs because a thread attempts to acquire the same lock twice without releasing it in between. Moreover, Java [AG96] provides the 'wait' and 'notify' mechanism as a means for a thread to temporarily release a lock, to hand it over to another thread, and to claim it back at a later stage. Locking and unlocking for synchronized methods, and the temporary release of a lock with the 'wait' and 'notify' mechanism, will be explained in this section.

To begin with, we extend the thread algebra developed so far with polarized termination constants to support class method calls.[6] After that, we extend it further such that classes with class methods can be treated at the level of threads. Finally, based on the foregoing, we give an explanation of the kind of synchronization mechanisms on which Java multi-threading is based: synchronized methods and the 'wait' and 'notify' mechanism. This explanation is given in a way consistent with each of the previously discussed strategic interleaving operators that can deal with blocking actions.

8.1. Polarized termination constants

In order to make it possible that threads are callable from other threads, thread algebra is extended with two additional constants: $\mathsf{S}+$ and $\mathsf{S}-$. These polarized termination constants represent termination with a positive result

[5] The precise way of dealing with local services has obvious consequences for deadlock behavior and it is an issue which is not so easily dealt with in the most general way. As such it constitutes an issue where the strategy dependence of deadlock behavior is apparent and where at the same time it is not obvious how to define deadlock in a way that is entirely independent of the strategy.

[6] Class methods belong to a class, whereas instance methods belong to an instance of a class. Instance methods are not covered in this paper. Class methods are often referred to as static methods.

Table 28. Axioms for polarized termination

$x \trianglelefteq \mathsf{S} \trianglerighteq y = \mathsf{S}$	PT1
$x \trianglelefteq \mathsf{S}+ \trianglerighteq y = x$	PT2
$x \trianglelefteq \mathsf{S}- \trianglerighteq y = y$	PT3
$x \trianglelefteq \mathsf{D} \trianglerighteq y = \mathsf{D}$	PT4
$x \trianglelefteq (u \trianglelefteq a \trianglerighteq v) \trianglerighteq y = (x \trianglelefteq u \trianglerighteq y) \trianglelefteq a \trianglerighteq (x \trianglelefteq v \trianglerighteq y)$	PT5

and termination with a negative result, respectively. Postconditional composition is now extended by allowing the middle argument to be an entire thread. The axioms for polarized termination are given in Table 28. The usefulness of $\mathsf{S}+$ and $\mathsf{S}-$ in treating classes with class methods at the level of threads will be demonstrated in the next subsection.

8.2. Classes as named collections of named threads

The term 'method' will now be used with its OO programming connotation, which should be distinguished from its role in the context of services. Both uses have in common, however, that they represent a request for which a Boolean reply is expected.

A class C is given as a finite collection of pairs of a class method name m and a thread p, where p is supposed always to end in a polarized termination or in D. A class description has the form:

$$\begin{aligned}&\mathit{class}{:}C = \{\\ &\quad m_1 = p_1,\\ &\quad \ldots\\ &\quad m_k = p_k\\ &\quad \}\end{aligned}$$

The class method names m_i that occur in this listing constitute the class method interface of the class C. Having available this very simple class notation, the thread algebra notation may be extended with class method calls by means of actions of the form $C..m$. It is of course admitted that such actions occur inside the threads p_i as well. The semantics of processes with these actions is given by the following equation, which allows a recursive removal of all class method calls in favor of postconditional composition on threads given a context that provides a thread for each method name in each class method interface:

$$x \trianglelefteq C..m_i \trianglerighteq y = x \trianglelefteq p_i \trianglerighteq y \ .$$

There is no plausible meaning in the case where no definition of a method m is available in the context, though D may be a reasonable choice. That case must be excluded in advance by means of conditions on thread descriptions, or stated differently by means of class type checking techniques. We will use combined definitions of a number of classes and a process that may involve class method calls on these classes. As a first example consider the definition

$$\begin{aligned}&\mathit{class}{:}G = \{\\ &\quad m1 = \mathsf{S}+ \trianglelefteq g.m1 \trianglerighteq \mathsf{S}-,\\ &\quad m2 = \mathsf{S}+ \trianglelefteq g.m2 \trianglerighteq \mathsf{S}-,\\ &\quad m3 = \mathsf{S}+ \trianglelefteq g.m3 \trianglerighteq \mathsf{S}-\\ &\quad \}\\ &\mathit{class}{:}H = \{\\ &\quad m1 = \mathsf{S}+ \trianglelefteq h.m1 \trianglerighteq \mathsf{S}-,\\ &\quad m3 = \mathsf{S}+ \trianglelefteq h.m3 \trianglerighteq \mathsf{S}-,\\ &\quad \}\\ &P = G..m2 \circ (\mathsf{D} \trianglelefteq G..m1 \trianglerighteq (H..m3 \circ \mathsf{S}))\ .\end{aligned}$$

Then one may derive:

$$
\begin{aligned}
& P \\
&= (\mathsf{S}{+}\trianglelefteq\ g.m2\ \trianglerighteq\mathsf{S}{-}) \circ (\mathsf{D}\trianglelefteq\ G..m1\ \trianglerighteq(H..m3 \circ \mathsf{S})) \\
&= g.m2 \circ (\mathsf{D}\trianglelefteq\ G..m1\ \trianglerighteq(H..m3 \circ \mathsf{S})) \\
&= g.m2 \circ (\mathsf{D}\trianglelefteq\ (\mathsf{S}{+}\trianglelefteq\ g.m1\ \trianglerighteq\mathsf{S}{-})\ \trianglerighteq(H..m3 \circ \mathsf{S})) \\
&= g.m2 \circ (\mathsf{D}\trianglelefteq\ g.m1\ \trianglerighteq(H..m3 \circ \mathsf{S})) \\
&= g.m2 \circ (\mathsf{D}\trianglelefteq\ g.m1\ \trianglerighteq((\mathsf{S}{+}\trianglelefteq\ h.m3\ \trianglerighteq\mathsf{S}{-}) \circ \mathsf{S})) \\
&= g.m2 \circ (\mathsf{D}\trianglelefteq\ g.m1\ \trianglerighteq(h.m3 \circ \mathsf{S}))\ .
\end{aligned}
$$

In the example above, all class methods have the same simple pattern. However, different and more complicated patterns are allowed. The only restriction is that all class methods always end in S+, S− or D. As a second example consider the following definition:

$$
\begin{aligned}
& \mathit{class}{:}Ca = \{ \\
& \quad m1 = f.u1 \circ f.u2 \circ Ca..m2 \circ \mathsf{S}{+}, \\
& \quad m2 = \mathsf{S}{+}\trianglelefteq\ g.v1\ \trianglerighteq\mathsf{S}{-}, \\
& \quad m3 = f.u3 \circ f.u4 \circ \mathsf{S}{+} \\
& \quad \} \\
& \mathit{class}{:}Cb = \{ \\
& \quad m1 = h.w1 \circ (\mathsf{S}{+}\trianglelefteq\ h.w2\ \trianglerighteq\mathsf{S}{-}), \\
& \quad m2 = \mathsf{S}{-}\trianglelefteq\ Ca..m2\ \trianglerighteq\mathsf{S}{+} \\
& \quad \} \\
& P = h.w0 \circ (\mathsf{D}\trianglelefteq\ Cb..m1\ \trianglerighteq(Ca..m3 \circ \mathsf{S}))\ .
\end{aligned}
$$

In this case one may derive:

$$
\begin{aligned}
& P \\
&= h.w0 \circ (\mathsf{D}\trianglelefteq\ h.w1 \circ (\mathsf{S}{+}\trianglelefteq\ h.w2\ \trianglerighteq\mathsf{S}{-})\ \trianglerighteq(Ca..m3 \circ \mathsf{S})) \\
&= h.w0 \circ h.w1 \circ (\mathsf{D}\trianglelefteq\ h.w2\ \trianglerighteq(Ca..m3 \circ \mathsf{S})) \\
&= h.w0 \circ h.w1 \circ (\mathsf{D}\trianglelefteq\ h.w2\ \trianglerighteq((f.u3 \circ f.u4 \circ \mathsf{S}{+}) \circ \mathsf{S})) \\
&= h.w0 \circ h.w1 \circ (\mathsf{D}\trianglelefteq\ h.w2\ \trianglerighteq(f.u3 \circ f.u4 \circ \mathsf{S}))\ .
\end{aligned}
$$

The two examples above illustrate that class methods may be considered a generalization of service methods.

8.3. Synchronized methods

The class definition syntax is now adapted by allowing the optional keyword *synchronized*, enclosed in parentheses, to precede the definition of a class method. To understand this feature, it is assumed that for each class C there is a unique service *classlock*:C which will provide a lock on that class. If a synchronized class method is called, first the lock on the relevant class is acquired by the thread calling the class method, and at the end of the execution of the body of the method this lock is released. The following example features a thread vector rather than a single thread. Indeed synchronized methods are only relevant in the context of multi-threading.

Table 29. Axioms for lock administration operators

$\mathsf{LA}^C(x) = \mathsf{LA}^{C,0}(x)$	LA1
$\mathsf{LA}^{C,n}(\mathsf{S}) = \mathsf{S}$	LA2
$\mathsf{LA}^{C,n}(\mathsf{S}+) = \mathsf{S}+$	LA2p
$\mathsf{LA}^{C,n}(\mathsf{S}-) = \mathsf{S}-$	LA2n
$\mathsf{LA}^{C,n}(\mathsf{D}) = \mathsf{D}$	LA3
$f \neq \mathit{classlock}{:}C \Rightarrow \mathsf{LA}^{C,n}(x \trianglelefteq f.m \trianglerighteq y) = \mathsf{LA}^{C,n}(x) \trianglelefteq f.m \trianglerighteq \mathsf{LA}^{C,n}(y)$	LA4
$\mathsf{LA}^{C,0}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{lock} \trianglerighteq y) = \mathit{classlock}{:}C.\mathit{lock} \circ \mathsf{LA}^{C,1}(x)$	LA5a
$\mathsf{LA}^{C,n+1}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{lock} \trianglerighteq y) = \mathsf{tau} \circ \mathsf{LA}^{C,n+2}(x)$	LA5b
$\mathsf{LA}^{C,0}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{unlock} \trianglerighteq y) = \mathsf{D}$	LA6a
$\mathsf{LA}^{C,1}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{unlock} \trianglerighteq y) = \mathit{classlock}{:}C.\mathit{unlock} \circ \mathsf{LA}^{C,0}(x)$	LA6b
$\mathsf{LA}^{C,n+2}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{unlock} \trianglerighteq y) = \mathsf{tau} \circ \mathsf{LA}^{C,n+1}(x)$	LA6c

$$
\begin{aligned}
&\mathit{class}{:}Ca = \{\\
&\quad m1 = f.u1 \circ f.u2 \circ Ca..m2 \circ \mathsf{S}+,\\
&\quad (\mathit{synchronized})\, m2 = \mathsf{S}+ \trianglelefteq\ g.v1\ \trianglerighteq \mathsf{S}-,\\
&\quad m3 = f.u3 \circ f.u4 \circ \mathsf{S}+\\
&\quad \}\\
&\mathit{class}{:}Cb = \{\\
&\quad (\mathit{synchronized})\, m1 = h.w1 \circ (\mathsf{S}+ \trianglelefteq\ h.w2\ \trianglerighteq \mathsf{S}-),\\
&\quad m2 = \mathsf{S}- \trianglelefteq\ Ca..m2\ \trianglerighteq \mathsf{S}+\\
&\quad \}\\
&P = \langle h.w0 \circ (\mathsf{D} \trianglelefteq\ Cb..m1\ \trianglerighteq (Ca..m3 \circ \mathsf{S}))\rangle \frown \langle Ca..m2 \circ \mathsf{S}\rangle\ .
\end{aligned}
$$

The defining equation for synchronized method calls (i.e. calls of methods that have the 'synchronized' keyword preceding their defining equation), is:

$$x \trianglelefteq\ C..m_i\ \trianglerighteq y = (\mathit{classlock}{:}C.\mathit{unlock} \circ x) \trianglelefteq\ (\mathit{classlock}{:}C.\mathit{lock} \circ p_i)\ \trianglerighteq (\mathit{classlock}{:}C.\mathit{unlock} \circ y)\ .$$

8.4. Lock administration operators

Unfortunately, the semantic description of synchronized class methods given above falls short of giving an account of synchronized class methods in Java because the same class may be locked more than once in the case of recursion. Only after as many unlockings as lockings is the lock in fact released. To take this into account, threads are transformed by lock administration operators ($\mathsf{LA}^-(\_)$) before being put in the thread vector. For each class used in a thread, a different lock administration operator is required. Each lock administration operator carries a count of the number of times that the class concerned is locked, thus making sure that the class lock is acquired only once and subsequent lockings as well as unlockings of it are disregarded till the unlocking corresponding to the locking by which the class lock was acquired. The lock administration operators are defined in Table 29. The use of the lock administration operators is exemplified by the adaptation of the example of Sect. 8.3 where the thread vector P is given by:

$$P = \langle \mathsf{LA}^{Ca}(\mathsf{LA}^{Cb}(h.w0 \circ (\mathsf{D} \trianglelefteq\ Cb..n1\ \trianglerighteq (Ca..m3 \circ \mathsf{S}))))\rangle \frown \langle \mathsf{LA}^{Ca}(\mathsf{LA}^{Cb}(Ca..m2 \circ \mathsf{S}))\rangle\ .$$

8.5. Wait and notify

Java's synchronization primitives include 'wait' and 'notify', which can be applied to locks on objects. A version of 'wait' that works for locks on classes in the absence of 'notify' may be written as $C..wait$ with the defining equation:

$$C..wait = classlock{:}C.wait \circ classlock{:}C.unwait \ .$$

Here 'wait' and 'unwait' are methods having the same effect on the lock as 'unlock' and 'lock', respectively. The reason for introducing synonyms for 'lock' and 'unlock' is that the methods introduced by expanding $C..wait$ should not be touched by the lock administration operators. This definition of 'wait' allows a thread to release a lock temporarily while keeping track of the nesting of its locking and unlocking attempts. The appropriate use of this definition of 'wait' is that it is used as a substitution operator $\mathsf{ELIM}_{wait}(\_)$ replacing '$C..wait$' by '$classlock{:}C.wait \circ classlock{:}C.unwait$'.

In the absence of the action 'notify', this modeling of 'wait' is reasonable. Notification, however, introduces another aspect. Consider the situation in which a thread has acquired the lock on a class while the lock is temporarily released by another thread by means of a 'wait' action for this lock. Then the latter thread will get back the lock when the former thread releases the lock, provided the former thread has performed a 'notify' action for this lock while it kept the lock.

It is assumed that the lock on a class C represents some condition ϕ^C which must hold just before the lock is acquired by any thread. Stated differently, the purpose of the locking mechanism is to ensure the truth of that condition at the time that a thread acquires the lock. Each thread blocked by that lock is waiting for the condition ϕ^C to hold. A 'notify' action for the lock on C gives the message to 'the system' that it is ensured that the condition ϕ^C holds. After a thread has performed this action, it may release the lock on C. After the lock on C has been released in this particular way, the interleaving strategy must ensure that one of the waiting threads is permitted to acquire this lock, thus making sure that the condition ϕ^C holds when the lock is acquired by the waiting thread. A thread performing a 'notify' action, need not immediately release the lock thereafter. Instead it is permitted to perform subsequent steps under the assumption that the condition ϕ^C is an invariant for each of these steps until the lock is released.

In order to model notification the classlock service becomes a rather involved service with an infinite state space. Instead of two states ($classlock{:}C(locked)$ and $classlock{:}C(unlocked)$), there are for each $n{\geq}0$ states:

- $classlock{:}C(locked, n)$,
- $classlock{:}C(unlocked, n)$,
- $classlock{:}C(lockedAndNotified, n)$, and
- $classlock{:}C(unlockedAndNotified, n)$.

'$C..wait$' is translated into '$classlock{:}C.wait \circ classlock{:}C.unwait$', and the action '$C..notify$' is translated into '$classlock{:}C.notify$'. All methods of the service '$classlock{:}C$' return T when performed. What matters is which methods are blocked, as the only influence of the service '$classlock{:}C$' is to force threads to wait by being blocked under certain circumstances. Here is a survey of the transitions of '$classlock{:}C$'.

unlocked, $n = 0$. This state is the initial state, and the only enabled method in this state is

- '$lock$' leading to the state $classlock{:}C(locked, 0)$.

locked, $n = 0$. The enabled methods from this state are

- '$unlock$' which brings it in the state $classlock{:}C(unlocked, 0)$;
- '$wait$' bringing it into state $classlock{:}C(unlocked, 1)$.

unlocked, $n > 0$. The only enabled action is

- '$lock$' which brings it in state $classlock{:}C(locked, n)$.

locked, $n > 0$. This state admits three methods:

- '$unlock$' which brings it in the state $classlock{:}C(unlocked, n)$;
- '$notify$' which brings it in the state $classlock{:}C(lockedAndNotified, n)$;

- '*wait*' bringing it into state $classlock{:}C(unlocked, n+1)$.

The method '*wait*' takes place if a thread keeping the lock on class C performs '$classlock{:}C.wait$' thus becoming an additional waiting thread. The class lock for C can now be acquired by any other thread irrespective of whether or not it will produce a notification before releasing the lock. In practical cases, if the new thread acquiring the lock performs a wait itself, it will probably do so because it finds condition ϕ^C not satisfied thus releasing the lock while becoming an additional waiting thread without producing a notification.

lockedAndNotified, $n > 0$. Now the two enabled methods are

- '$notify$' which will not change the state;
- '*unlock*' is which leads to the state $classlock{:}C(unlockedAndNotified, n)$.

Successive notifications may 'get lost' because '$notify$' leaves the state unchanged.

unlockedAndNotified, $n > 0$. From this state the only method is

- '*unwait*' is enabled which leads to state $classlock{:}C(locked, n-1)$.

This takes place if after a notification one of the waiting threads (which one is to be determined either explicitly of implicitly by the particular interleaving strategy at hand) regains the lock thus reducing the number of waiting threads by one.

The service '$classlock{:}C$' thus specified can be used in connection with any of the previously specified interleaving strategies, provided these can deal with blocking actions. It should be used in combination with the lock administration operators.

In this way, a reasonably formal and accurate explanation of some of the Java synchronization primitives has been obtained. The purpose of our description of 'wait' and 'notify' is not to write a fully precise Java semantics, but to provide a simple introduction to and explanation of the kind of thread synchronization mechanisms on which Java multi-threading is based.

9. Multi-threading and multi-processing

In this section, we outline in brief how thread algebra can be extended such that systems consisting of different multi-threaded programs on the same machine are covered.

A process in the sense of conventional operating system terminology may be identified with a multi-thread. In order to prevent confusion with the terminology from process theory, a process of this kind will be termed a system process. In the case of multi-processing, a single execution architecture may involve several system processes. Thus multi-processing refers to the concurrent existence of different system processes rather than to the concurrent existence of different threads in a single system process.

The complexity of describing multi-processing in thread algebra arises from the fact that different multi-threads may interact with the same components. Unix sockets are shared components for different system processes. At the level of thread algebra, sockets represent target services for the threads in a single multi-thread, whereas at system level sockets have a purely auxiliary nature.

Deadlock is to be defined per multi-thread as before, where the para-target services are only those components for which the use is not shared with other multi-threads. However, models of multi-processing cannot be formalized as easily as models of multi-threading, even if a very simplistic viewpoint towards strategic interleaving is adopted. In the case of multi-threads, it is important to model some vital operating system activity, including at least: starting a process from data containing a description of a startup thread (e.g. a program for it), terminating a process by a forced quit, monitoring termination and deadlock and providing options for proceeding thereafter.

If all of these matters are ignored, it is reasonable to model the cooperation of several system processes by pure cyclic strategic interleaving (as described in Sect. 3.1) of the various multi-threads per system process. The multi-threads may be placed in a vector by ordering them alphabetically on process names. Variations regarding this ordering, step counting, a variable number of steps depending on the process name or a process priority are reasonable. Thus, when describing a multi-processing execution architecture in the simplest setting, two strategic interleaving operators are needed: a local one for the thread vectors (assuming that single system processes admit multi-threading) and a global one (at the level of the machine) operator for interleaving the various multi-threads that denote system processes. The result of a strategic interleaving of several multi-threads may be termed a multi-system process. In [BM05b, BM06c, BM07b], we embroider on this theme.

10. Conclusions

We have outlined an algebraic theory about threads and multi-threading based on strategic interleaving. Interleaving operators for a number of plausible deterministic interleaving strategies have been specified in a simple and concise way. With that, we have actually demonstrated that it is essentially open-ended what counts as an interleaving strategy. The theory has been developed to the level of detail needed to capture a reasonable definition of deadlock. We have used deadlock as an example of a property of multi-threaded programs that depends on the interleaving strategy used.

The claim is put forward that the basic intuitions concerning deadlock in the setting of programming with a program notation for multi-threading exist at the level of abstraction implicit in the polarized process algebra model of threads. As a consequence, a deeper understanding of deadlock and deadlock freedom robust against a very wide choice of interleaving strategies is not considered a part of those intuitions. Whereas we have succeeded in the specification of a number of plausible interleaving strategies that demonstrate various key features concerning the cooperation of threads, we have not developed a theory that easily allows one either to develop the most general notion of an interleaving strategy or to develop a clear picture of the collection of all possible correct interleaving operators.

Of course both developments are possible. However, we contend that in both approaches one will engage in a train of thought not immediately helpful for a programmer, whereas the theory outlined in this paper seems to convey no information that might be considered redundant for a programmer who wants to acquire a first intuition of multi-threading by reading a self-contained theoretical paper rather than by experimenting with a program notation and a system implementing the execution of the programs written in the program notation concerned.

Thread algebra is a theory about threads and multi-threading intended to be useful for (i) gaining insight into the semantic issues concerning the multi-threading related features found in contemporary object-oriented programming languages such as Java and C#, and (ii) simplified formal description of multi-threaded programs and verification of general properties of those programs. In [BM06d], we extend the theory developed in this paper with features that allow for details of multi-threading that come up where it is intertwined with object-orientation to be dealt with. In ongoing work, we are extending the theory developed in this paper with features that allow for details of multi-threading that come up where threads are distributed over the nodes of a network to be dealt with. Those features include explicit thread migration and load balancing.

Our work on thread algebra exceeds the domain of single multi-threaded programs. In [BM05b, BM06c], we extend the theory developed in this paper with features to cover systems that consist of several multi-threaded programs on various hosts in different networks. Our work on a formal approach to design new micro-architectures, see [BM07a, BM05a, BM06a, BM06b], is a notable application of thread algebra. The approach is based on Maurer's model for computers [Mau66, Mau06], thread algebra and program algebra [BL02].

Options for future work include:

- formalization of mechanisms found in contemporary programming languages, such as Java pipes, using the theory developed in this paper;
- extension of program algebra [BL02] with features for multi-threading, with a semantics based on the theory developed in this paper.

Acknowledgements

We thank Alban Ponse and Mark van der Zwaag from the University of Amsterdam, Programming Research Group, for pointing out a shortcoming of the axioms for cyclic interleaving with blocked actions and memory presented in a draft of this paper and for suggesting improvements of the semantics of fork instructions in PGLC presented in a draft of this paper. Moreover, we thank two referees for valuable comments concerning the presentation of the paper.

References

[AFV01] Aceto L, Fokkink WJ, Verhoef C (2001) Structural operational semantics. In: Bergstra JA, Ponse A, Smolka SA (eds) Handbook of process algebra. Elsevier, Amsterdam, pp. 197–292

[AG96] Arnold K, Gosling J (1996) The Java programming language. Addison-Wesley, Reading, MA

[BB92] Baeten JCM, Bergstra JA (1992) Process algebra with signals and conditions. In: Broy M (ed) Programming and mathematical methods, Vol. F88 of NATO ASI Series. Springer, Heidelberg, pp. 273–323
[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. Lecture notes in computer science, Vol 2719. Springer, Heidelberg, pp. 1–21
[BB05] Bergstra JA, Bethke I (2005) Polarized process algebra with reactive composition. Theor Comput Sci 343:285–304
[BBP05] Bergstra JA, Bethke I, Ponse A (2005) Decision problems for pushdown threads. Report PRG0502, Programming Research Group, University of Amsterdam
[BHR84] Brookes SD, Hoare CAR, Roscoe AW (1984) A theory of communicating sequential processes. J ACM 31(3):560–599
[BK84] Bergstra JA, Klop JW (1984) Process algebra for synchronous communication. Inform Control 60(1/3):109–137
[BL02] Bergstra JA, Loots ME (2002) Program algebra for sequential code. J Logic Algebraic Programm 51(2):125–156
[BM05a] Bergstra JA, Middelburg CA (2005) Simulating Turing machines on Maurer machines. Computer Science Report 05-28, Department of Mathematics and Computer Science, Eindhoven University of Technology, November 2005
[BM05b] Bergstra JA, Middelburg CA (2005) A thread algebra with multi-level strategic interleaving. In: Cooper SB, Löwe B, Torenvliet L (eds) CiE 2005. Lecture notes in computer science, Vol 3526. Springer, Heidelberg, pp. 35–48
[BM06a] Bergstra JA, Middelburg CA (2006) Maurer computers for pipelined instruction processing. Computer Science Report 06-12, Department of Mathematics and Computer Science, Eindhoven University of Technology, March 2006
[BM06b] Bergstra JA, Middelburg CA (2006) Synchronous cooperation for explicit multi-threading. Computer Science Report 06-29, Department of Mathematics and Computer Science, Eindhoven University of Technology, September 2006
[BM06c] Bergstra JA, Middelburg CA (2006) Thread algebra with multi-level strategies. Fundam Informa 71(2/3):153–182
[BM06d] Bergstra JA, Middelburg CA (2006) A thread calculus with molecular dynamics. Computer Science Report 06-24, Department of Mathematics and Computer Science, Eindhoven University of Technology, August 2006
[BM07a] Bergstra JA, Middelburg CA (2007) Maurer computers with single-thread control. Fundam Inform (in press). Preliminary version: Computer Science Report 05-17, Department of Mathematics and Computer Science, Eindhoven University of Technology
[BM07b] Bergstra JA, Middelburg CA (2007) A thread algebra with multi-level strategic interleaving. Theory Comput Syst 41(1) (in press). Preliminary version: Computer Science Report 06-28, Department of Mathematics and Computer Science, Eindhoven University of Technology
[BP02] Bergstra JA, Ponse A (2002) Combining programs and state machines. J Logic Algebraic Program 51(2):175–192
[BP07] Bergstra JA, Ponse A (2007) Execution architectures for program algebra. J Appl Logic (in press). Preliminary version: Logic Group Preprint Series 230, Department of Philosophy, Utrecht University
[BW90] Baeten JCM, Weijland WP (1990) Process algebra. Cambridge tracts in theoretical computer science, Vol 18. Cambridge University Press, Cambridge
[GJSB00] Gosling J, Joy B, Steele G, Bracha G (2000) The Java language specification, 2nd edn. Addison-Wesley, Reading, MA
[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Englewood Cliffs
[HWG03] Hejlsberg A, Wiltamuth S, Golde P (2003) C# Language Specification. Addison-Wesley, Reading, MA
[JL00] Jesshope CR, Luo B (2000) Micro-threading: A new approach to future RISC. In: Australian computer architecture conference 2000. IEEE Computer Society Press, pp. 34–41
[Mau66] Maurer WD (1966) A theory of computer instructions. J ACM 13(2):226–235
[Mau06] Maurer WD (2006) A theory of computer instructions. Sci Comput Program 60:244–273
[Mid03] Middelburg CA (2003) An alternative formulation of operational conservativity with binding terms. J Logic Algebraic Program 55(1/2):1–19
[Mil80] Milner R (1980) A calculus of communicating systems. Lecture notes in computer science, Vol 92. Springer, Berlin
[Mil89] Milner R (1989) Communication and concurrency. Prentice-Hall, Englewood Cliffs
[MRG05] Mousavi MR, Reniers MA, Groote JF (2005) Notions of bisimulation and congruence formats for SOS with data. Inform Comput 200:107–147
[Pet62] Petri CA (1962) Kommunikation mit Automaten. PhD thesis, Institut für Instrumentelle Mathematik, Bonn
[Rei85] Reisig W (1985) Petri nets: an introduction. Monographs in theoretical computer science, Vol. 4. An EATCS Series. Springer, Berlin
[vG87] van Glabbeek RJ (1987) Bounded nondeterminism and the approximation induction principle in process algebra. In: Brandenburg FJ, Vidal-Naquet G, Wirsing M (eds) STACS 87. Lecture notes in computer science, Vol 247. Springer, Heidelberg, pp. 336–347

Received 23 November 2004
Revised 22 December 2006
Accepted 18 January 2007 by C. B. Jones
Published online 22 February 2007