Thread Algebra for Strategic Interleaving

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

[1] Programming Research Group, University of Amsterdam,
P.O. Box 41882, 1009 DB Amsterdam, the Netherlands
janb@science.uva.nl
[2] Department of Philosophy, Utrecht University,
P.O. Box 80126, 3508 TC Utrecht, the Netherlands
janb@phil.uu.nl
[3] Computing Science Department, Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
keesm@win.tue.nl

Abstract. We present an extension of the polarized process algebra BPPA, an algebraic theory about sequential program behaviors. The extension is called thread algebra and is proposed as a tool for the description and analysis of multi-threaded program behaviors. Strategic interleaving refers to the form of concurrency where some interleaving strategy is used rather than arbitrary interleaving. Strategic interleaving is considered characteristic of multi-threading. Multi-threaded concurrency is more limited than general concurrency based on arbitrary interleaving.

Keywords: thread algebra, multi-threading, polarized process algebra, strategic interleaving operators, execution architectures, deadlock freedom.

1998 CR Categories: D.1.3, D.2.4, D.3.3, F.1.2, F.3.1.

1 Introduction

Theories about concurrent processes such as ACP [4], CCS [12], and CSP [9] 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 program notations such as C# [8] and Java [2], arbitrary interleaving is not the most useful intuition. A thread may be considered a process describing a program under execution. Multi-threading involves some form of parallel composition of different threads. Attempting to give an informal definition of a thread we arrive at a listing of keys aspects and properties: (i) athread is the behavior of a sequential program as run on a machine, (ii) at any time it has some form of unique identity, (iii) it will have a time of creation and its individual history thereafter, (iv) during its life it co-exists with zero or more other threads in some execution architecture, (v) its actions may affect the state of system components present in the architecture, and (vi) external observations of a thread are made indirectly via the behavior of system components which have been activated by the thread.

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 processes occur in some order where at each time only one step is taken either from the first or from the second process. Threads will be modeled as discrete behaviors, and each approach to their their parallel composition inherits from the theory of parallel composition of discrete behaviors.

Arbitrary Interleaving Arbitrary interleaving has been proposed by many authors as a plausible, general, if not idealized model of the operation of concurrent systems. 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 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.

True Concurrency In contrast to arbitrary interleaving concurrency so-called true concurrency models have been proposed which, in some cases better grasp the concept of parallelism. In true concurrency models one intends to avoid causal dependencies that may solely arise from the sequencing of activities given by any arbitrary interleaving of two behaviors. 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.

Strategic Interleaving Rather than assuming some mechanism of arbitrary interleaving it may be assumed that some deterministic interleaving strategy determines the ordering of performing actions from various threads. 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' to indicate a more constrained alternative to arbitrary interleaving in much the same way as true concurrency is its alternative in the less constrained direction.

Strategic interleaving is viewed by the authors as a modeling technique specifically aimed at questions arising in connection with multi-threaded programs and systems.

1.2 Thread Algebra versus Process Algebra

Thread algebra is an algebraic theory of behaviors specifically designed for the specification and analysis of 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 that place will take the form of the position of a thread in the so-called thread vector.

Thread algebra[1] is a design on top of the polarized process algebra that serves in [5, 3] as the semantic basis for a theory of sequential programs and their behavior. Polarized process algebra is far less general than ACP style process algebra and its design focuses specifically to the semantic analysis of sequential deterministic programs. The semantics of a sequential program is supposed to be a polarized process. Polarization in process algebra is understood in [5] along the axis of the client-server dichotomy. Basic actions in a polarized process are either request expecting a reply or service offerings promising a reply. Thread algebra may be viewed as client side polarized process algebra because all running programs and for that reason all threads are viewed as clients generating requests for their environment.

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 [6] the dichotomy has been given as program versus state machine. Here the program represents a thread, though less abstract and a state machine is a reactive component to which a program issues its instructions. This suggest 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 issues 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 which 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

[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.

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 for external parties. Printing a document, sending an email or showing information on a display are typical examples of calling a target service. Writing information to permanent memory is another example, because permanent memory may be observed from 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 which all commands involved have on the target services provided by the execution environment.

para-target services. Services that are not target services will be called para-target services. Alternative names might be auxiliary services, internal services and so on. We propose to use the phrase para-target in order to avoid any misleading connotations.

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 complete control of that thread. Local services split in local target services and local para-target services. Local para-target services exist to support a thread in creating useful, or at least intended, behavior towards (local or shared) target services.

shared services. A shared service in a multi-threaded system provides its service to all existing threads. Just as local services shared services can be distinguished in shared target services and shared para-target services.

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.

1.4 Thread Vectors and Strategic Interleaving

In order to deal with multi-threading it is assumed that the pool of threads of a system takes the form of a linearly ordered list, called the thread vector. Strategic interleaving operators describe interleaving strategies on thread vectors. These operators are the content of thread algebras. The main purpose of this paper is to specify a number of strategic interleaving operators that provide an understanding of how multi-threaded systems may be executed. 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, one might even say intended for multi-threading, can be specified in a concise and comprehensible

way. The family of strategic interleaving operators (SIOPs) grows by introducing more features for thread behavior and interaction, the simplest case being cyclic rotation. Features treated in this paper include thread creation, thread termination, action enabledness tests, and locking/unlocking of services.

By designing an incremental SIOP hierarchy 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

After the introduction of threads as polarized processes, thread vectors are introduced to represent the state of a multi-threaded system. A thread vector only takes into account the state of control of a multi-threaded system under execution. A SIOP now maps thread vectors on a single polarized process. A process obtained via a SIOP is called a multi-thread. A portfolio of SIOPs is displayed, all based on cyclic scheduling.

Then an informal account of the co-operation of threads and multi-threads with a combination of services is given. This leads to a family of examples demonstrating the the notion of a deadlock is sensitive to the ordering of a thread vector as well as to the particular interleaving strategy.

Thereafter a formalization is given of para-target services, and the interaction between threads and multi-threads and para-target services (both local and shared) is specified by means of the use-operator. A restricted from of commutativity and idempotence is found for the use-operator. Then a formal definition of a deadlock is given that validates the observations made earlier in a slightly more informal setting.

2 Basic Polarized Process Algebra BPPA

In this Section, we discuss BPPA, a form of process algebra which is tailored to the use for the description of sequential program behavior. In BPPA, it is assumed that there is a fixed but arbitrary finite set of *basic actions* A with $\mathsf{tau} \notin \mathsf{A}$. We write $\mathsf{A}_{\mathsf{tau}}$ for $\mathsf{A} \cup \{\mathsf{tau}\}$. BPPA has the following constants and operators:

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

[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 (evaluated conditions) is also more easily supported in thread algebra.

We introduce *action prefixing* as an abbreviation: $a \circ x$ abbreviates $x \trianglelefteq a \trianglerighteq x$. We use infix notation for postconditional composition. Viewing the basic action as an argument as well, we have a ternary postconditional composition operator $\_ \trianglelefteq \_ \trianglerighteq \_$, where the middle argument must be a basic action or tau. The special action tau will emerge as the result of calculations within the thread algebra.

2.1 Informal Explanation

The operational intuition behind this syntax is that each basic action represents a command which is to be processed by the execution environment of the process. More specifically a basic action is taken as a command for a service offered by the environment. The processing of a command may involve a change of state of this environment, or more specifically of a service provided by the environment. 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 under execution. The process $x \trianglelefteq a \trianglerighteq y$ will proceed as x if the processing of a leads to the reply T (called a positive reply) indicating the successful processing of a, and it will proceed as y if the processing of a leads to the reply F (called a negative reply) indicating the unsuccessful processing of a. The action tau plays a special role. Its execution will never change any state and always produce a positive reply. It is a concrete internal action. Here concrete is meant as the opposite of abstract which implies that its presence matters and there is no abstraction made of it via equations that remove these actions in some contexts. Its name 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.

2.2 CPO structure, Recursion and Continuity

Folllowing [3] a CPO structure can be imposed on the domain of BPPA. Then guarded recursion equations represent continuous operators having appropriate fixed points. These matters will not be repeated here, taking for granted that guarded systems of recursion equations allow one to define unique polarized processes. Guardedness is a requirement that guarantees that repeated substitution of the righthand sides of equations for the lefthand side variables eventually produces an expression of the form D, S or $P \trianglelefteq a \trianglerighteq Q$. As systems of equations become more involved, especially if infinite systems are considered, guardedness may become hard to define and it may also become undecidable. In the sequel of this paper all recursion equations may easily be classified as guarded, though a formal definition of guardedness covering all cases is not presented here.

2.3 Projective Limit Model

The projective limit characterization of process equivalence on polarized processes is based on the notion of a finite approximation of depth n. When for all

Table 1. Axioms for the operators $\pi_n(\_)$

$\pi_0(x) = \mathsf{D}$	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

n these approximations are identical for two given polarized processes, both processes are considered identical. This allows one to eliminate recursion in favor of the infinitary proof-rule AIP. Following [5] approximation of depth n is phrased in terms of a so-called projection operator $\pi_n(\_)$. The projection operators are defined inductively by means of the axioms in Table 1. In P3, a stands for an arbitrary action from $\mathsf{A}_{\mathsf{tau}}$.

2.4 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 depends 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 if its processing succeeds and a negative reply if its processing fails. Whether the processing of the command succeeds or fails 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 succeed, but not if the diskette turns out to be write-protected.

In the structural operational semantics, we represent an execution environment by a function $\rho : (\mathsf{A} \times \{\mathsf{T}, \mathsf{F}\})^* \to \mathcal{P}(\mathsf{A} \times \{\mathsf{T}, \mathsf{F}\})$ that satisfies the following condition: $(a, b) \notin \rho(\alpha) \Rightarrow \rho(\alpha \frown \langle (a, b) \rangle) = \emptyset$ for all $a \in \mathsf{A}$, $b \in \{\mathsf{T}, \mathsf{F}\}$, and $\alpha \in (\mathsf{A} \times \{\mathsf{T}, \mathsf{F}\})^*$. We write $\mathcal{E}$ for the set of all those functions.

Given an execution environment $\rho \in \mathcal{E}$ and an action $a \in \mathsf{A}$, the *derived* execution environment of ρ after processing a with *success*, 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 *failure*, 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 $a \in \mathsf{A}_{\mathsf{tau}}$ and $\rho, \rho' \in \mathcal{E}$;
a unary relation $\langle \_, \rho \rangle \downarrow$ for each $\rho \in \mathcal{E}$;
a unary relation $\langle \_, \rho \rangle \uparrow$ for each $\rho \in \mathcal{E}$.

Table 2. Structural operational semantics of BPPA

$$\frac{}{\langle \mathsf{S}, \rho\rangle\downarrow} \qquad \frac{}{\langle \mathsf{D}, \rho\rangle\uparrow}$$

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

$$\frac{}{\langle x \trianglelefteq a \trianglerighteq y, \rho\rangle\uparrow}\ (a, \mathsf{T}) \notin \rho(\langle\,\rangle), (a, \mathsf{F}) \notin \rho(\langle\,\rangle) \qquad \frac{}{\langle x \trianglelefteq \mathsf{tau} \trianglerighteq y, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x, \rho\rangle}$$

Table 3. Structural operational semantics for the operators $\pi_n(\_)$

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

Table 4. Structural operational semantics for the constants $\langle X|E\rangle$

$$\frac{\langle\langle t_X|E\rangle, \rho\rangle \xrightarrow{a} \langle x', \rho'\rangle}{\langle\langle X|E\rangle, \rho\rangle \xrightarrow{a} \langle x', \rho'\rangle}\ X = t_X \in E \qquad \frac{\langle\langle t_X|E\rangle, \rho\rangle\downarrow}{\langle\langle X|E\rangle, \rho\rangle\downarrow}\ X = t_X \in E \qquad \frac{\langle\langle t_X|E\rangle, \rho\rangle\uparrow}{\langle\langle X|E\rangle, \rho\rangle\uparrow}\ X = t_X \in E$$

The three kinds of transition relations are called the *action step*, *termination*, and *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 is capable of first performing action a and then proceeding as process t' in execution environment ρ';

$\langle t, \rho\rangle\downarrow$**:** in execution environment ρ, process t is capable of terminating successfully;

$\langle t, \rho\rangle\uparrow$**:** in execution environment ρ, process t is neither capable of performing an action nor capable of terminating successfully.

The structural operational semantics of BPPA is described by the transition rules given in Table 2 where a ranges over A. The structural operational semantics for the operators $\pi_n(\_)$ is described by the transition rules given in Table 3 with a ranging over $\mathsf{A}_{\mathsf{tau}}$.

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 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 4 (with a ranging over $\mathsf{A}_{\mathsf{tau}}$). 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 $\langle t_1, \rho\rangle\downarrow$, then $\langle t_2, \rho\rangle\downarrow$;
- if $B(t_1, t_2)$ and $\langle t_1, \rho\rangle\uparrow$, then $\langle t_2, \rho\rangle\uparrow$.

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 $B(t_1, t_2)$. If B is a bisimulation and $B(t_1, t_2)$, then we say that B is a bisimulation *witnessing* $t_1 \leftrightarrows t_2$.

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. [1]).

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 [13], bisimulation equivalence as defined above is *stateless* bisimulation equivalence and the intended variant is *initially stateless* bisimulation equivalence.

3 Thread Vectors and Multi-Threads

In [5] its has been outlined how and why polarized processes are a natural candidate for the specification of sequential program semantics. Assuming that a thread is a process representing a program being run it is reasonable to view all polarized processes as threads. A thread vector is a sequence of threads. Thread vectors are denoted as follows: $\langle\,\rangle$ for the empty sequence, $\langle x\rangle$ for a sequence of length one, and $\alpha \frown \beta$ for the concatenation of two sequences. We assume that the following identity holds: $\alpha \frown \langle\,\rangle = \langle\,\rangle \frown \alpha = \alpha$.

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. The main objective of thread algebra is to specify a collection of strategic interleaving operators, capturing some essential aspects of multi-threading.

Subscripts of strategic interleaving operators will be used to indicate features which are dealt with in addition to the minimum given by the cyclic interleaving operator $\|_{csi}(\_)$ introduced first. Superscripts will be used to encode state information when needed.

3.1 Strategic Interleaving by Cyclic Rotation

The axioms for cyclic interleaving ($\|_{csi}(\_)$) are given in Table 5. In CSI4, a stands

Table 5. Axioms for the strategic interleaving operator $\|_{csi}(\_)$

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

Table 6. Axioms for the operator $\mathsf{S_D}(\_)$

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

Table 7. Structural operational semantics for the operators $\|_{csi}(\_)$ and $\mathsf{S_D}(\_)$

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

$$\frac{\langle x_1,\rho\rangle\not\rightarrow,\ldots,\langle x_k,\rho\rangle\not\rightarrow,\langle x_l,\rho\rangle\uparrow,\langle x_{k+1},\rho\rangle \xrightarrow{a} \langle x'_{k+1},\rho'\rangle}{\langle\|_{csi}(\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha),\rho\rangle \xrightarrow{a} \langle\|_{csi}(\alpha \frown \langle\mathsf{D}\rangle \frown \langle x'_{k+1}\rangle),\rho'\rangle}\ (0 \le k < n, 0 < l \le k)$$

$$\frac{\langle x_1,\rho\rangle\downarrow,\ldots,\langle x_n,\rho\rangle\downarrow}{\langle\|_{csi}(\langle x_1\rangle \frown \ldots \frown \langle x_n\rangle),\rho\rangle\downarrow} \qquad \frac{\langle x_1,\rho\rangle\not\rightarrow,\ldots,\langle x_n,\rho\rangle\not\rightarrow,\langle x_l,\rho\rangle\uparrow}{\langle\|_{csi}(\langle x_1\rangle \frown \ldots \frown \langle x_n\rangle),\rho\rangle\uparrow}\ (0 < l \le n)$$

$$\frac{\langle x,\rho\rangle \xrightarrow{a} \langle x',\rho'\rangle}{\langle\mathsf{S_D}(x),\rho\rangle \xrightarrow{a} \langle\mathsf{S_D}(x'),\rho'\rangle} \qquad \frac{\langle x,\rho\rangle\downarrow}{\langle\mathsf{S_D}(x),\rho\rangle\uparrow} \qquad \frac{\langle x,\rho\rangle\uparrow}{\langle\mathsf{S_D}(x),\rho\rangle\uparrow}$$

for an arbitrary action from $\mathsf{A_{tau}}$. In CSI3, the auxiliary *deadlock at termination* operator $\mathsf{S_D}(\_)$ is used. This operator turns termination into deadlock. Its axioms appear in Table 6. In S2D3, a stands for an arbitrary action from $\mathsf{A_{tau}}$.

The structural operational semantics for the operators $\|_{csi}(\_)$ and $\mathsf{S_D}(\_)$ is described by the transition rules given in Table 7. Here $\langle x,\rho\rangle\not\rightarrow$ stands for the set of all negative conditions $\neg(\langle x,\rho\rangle \xrightarrow{a} \langle t',\rho'\rangle)$ where t' is a closed term of BPPA, $\rho' \in \mathcal{E}$ and $a \in \mathsf{A}$.

Bisimulation equivalence is also a congruence with respect to the operators $\|_{csi}(\_)$ and $\mathsf{S_D}(\_)$. 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. [11]).

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 moderately till considerably more involved, and it explicit presentation adds at most marginally to a better understanding of the interleaving strategy concerned.

Table 8. Axioms for the strategic interleaving operator $\|^{W2}_{csi}(\_)$

$\|^{W2}_{csi}(\langle\,\rangle) = \mathsf{S}$	CSIW1
$\|^{W2}_{csi}(\langle\mathsf{S}\rangle \frown \alpha) = \|^{W2}_{csi}(\alpha)$	CSIW2
$\|^{W2}_{csi}(\langle\mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\|^{W2}_{csi}(\alpha))$	CSIW3
$\|^{W2}_{csi}(\langle x \trianglelefteq a \trianglerighteq y\rangle) = \|^{W2}_{csi}(\langle x\rangle) \trianglelefteq a \trianglerighteq \|^{W2}_{csi}(\langle y\rangle)$	CSIW4
$\|^{W2}_{csi}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \langle u \trianglelefteq b \trianglerighteq v\rangle \frown \alpha) =$ $a \mid b \circ \|^{W2}_{csi}(\alpha \frown \langle x\rangle \frown \langle u\rangle)$ $\triangleleft\, a\#b \,\triangleright$ $\|^{W2}_{csi}\,(\langle u \trianglelefteq b \trianglerighteq v\rangle \frown \alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|^{W2}_{csi}(\langle u \trianglelefteq b \trianglerighteq v\rangle \frown \alpha \frown \langle y\rangle)$	CSIW5

3.2 Basic Action Width Two and Beyond

The equations from Table 5 exclude 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 $\|_{csi}(\_)$ provides basic action width one only. 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 any 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 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. [10]. In order to specify a strategic interleaving operator for width 2 it is a reasonable simplification to assume that basic actions independent of other basic actions always return T. Moreover the simultaneous execution of two independent basic actions generates a positive return value in all circumstances. Thus, if an action may return both results it cannot be performed simultaneously with any other basic action. A strategy with action width 2 is presented in Table 8. In CSIW4 and CSIW5, a and b stand for arbitrary actions from $\mathsf{A}_{\mathsf{tau}}$. A similar but more complicated axiomatization can be found for higher widths of course. The remainder of this paper focusses on the case of basic action width one which is vital for an understanding of multi-thread computer programming, leaving a development in the direction of processor architecture for future elaboration.

As an auxiliary operator use is made of the conditional operator $\_ \triangleleft \_ \triangleright \_$, where the second argument must be a boolean value. For each boolean value a defining equation is needed: $x \triangleleft \mathsf{T} \triangleright y = x$ and $x \triangleleft \mathsf{F} \triangleright y = y$.

3.3 Step Counting

A simple variation of $\|_{csi}(\_)$ is $\|^{k,l}_{csi}(\_)$ which is equipped with counters and gives each thread a fixed number k of consecutive turns. The superscript l indicates

Table 9. Axioms for the strategic interleaving operator $\|^{k,l}_{csi}(\_)$

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

Table 10. Additional axiom for $\|^{k,l}_{csi}(\_)$ with yield action

$\|^{k,l}_{csi}(\langle x \trianglelefteq \mathsf{YIELD} \trianglerighteq y\rangle \curvearrowright \alpha) =$	
$\mathsf{tau} \circ (\|^{k,1}_{csi}(\alpha \curvearrowright \langle x\rangle) \triangleleft \alpha \neq \langle\,\rangle \triangleright \|^{k,l+1}_{csi}(\langle y\rangle))$	CSIscY

that $l-1$ of the k steps have already been performed. Its axioms are given in Table 9. In CSIsc4, a stands for an arbitrary action from $\mathsf{A}_{\mathsf{tau}}$. CSIsc0 defines an additional operator: $\|^{1}_{csi}(\_)$. Clearly for all x, $\|_{csi}(x) = \|^{1}_{csi}(x)$. The advantage of this interleaving strategy is that fewer context switches, i.e. moves from one thread to another one are scheduled, which may in some cases speed up the execution of a system.

An Action YIELD Yielding within a thread stands for handing over control to another thread. This becomes meaningful in the step counting strategy with $k > 1$. In Table 10 an axiom for a yield action is given. Recall that the special action tau serves as an internal action.

3.4 Current Thread Persistence

Having available an action YIELD or any other action that invokes rotation of the thread vector, cyclic rotation may be dropped in favor of rotations explicitly asked for by the thread. A strategy of this kind is said to provide current thread persistence, thus expressing that the current thread switches to another thread only when needed. The family of strategies that is outlined below will be based on cyclic rotation rather than on current thread persistence because some form of rotation taking place outside the control of the individual threads is considered essential for multi-threading at any rate. Table 11 provides axioms for a current thread persistent strategy.

3.5 Thread Creation or Forking

Forking off a thread is a step in the execution of some thread which gives rise to the creation of a new additional thread which will be running in the same

Table 11. Axioms for a strategy with current thread persistence $\|_{ctp}(\_)$

$\|_{ctp}(\langle\rangle) = \mathsf{S}$	ctpSI1
$\|_{ctp}(\langle\mathsf{S}\rangle \frown \alpha) = \|_{ctp}(\alpha)$	ctpSI2
$\|_{ctp}(\langle\mathsf{D}\rangle \frown \alpha) = \mathsf{S_D}(\|_{ctp}(\alpha))$	ctpSI3
$\|_{ctp}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) = \|_{ctp}(\langle x\rangle \frown \alpha) \trianglelefteq a \trianglerighteq \|_{ctp}(\langle y\rangle \frown \alpha)$	ctpSI4
$\|_{ctp}(\langle x \trianglelefteq \mathsf{YIELD} \trianglerighteq y\rangle \frown \alpha) = \mathsf{tau} \circ (\|_{ctp}(\alpha \frown \langle x\rangle) \triangleleft \alpha \neq \langle\rangle \triangleright \|_{ctp}(\langle y\rangle))$	ctpSIY

Table 12. Additional axiom for $\pi_n(\_)$ with fork actions

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

context. We intend to separate the action 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. This case distinction is returned as the result of a fork action to the thread which performed the fork, allowing it to make its further execution dependent on whether or not the fork actually succeeded. That may depend on a variety of aspects which are immaterial for the act of forking 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.

There are some axioms for the thread forking operator because it may appear inside recursive definitions of threads. To deal with that matter the projective limit model characterization of process identity on polarized processes will be used that easily carries over to this case.

The projection operators are extended inductively by means of the axiom in Table 12. The working of AIP in this case can be appreciated when considering a recursive process definition such as:

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

Then $\pi_n(P)$ is provably equal to a closed term for each n. The axioms for $\|_{csi,f}(\_)$ (cyclic interleaving with forking) are given in Table 13. In CSIf4, a stands for an arbitrary action from $\mathsf{A_{tau}}$ different from the actions of the form $\mathsf{NT}(x)$ and the action NT. Here, the additional basic action NT is used. Its processing succeeds if the creation of a new thread can take place and its processing fails if the creation of a new thread cannot take place. For instance, the thread vector may be run in an execution environment that constrains its length. Then NT will return T only if the number of threads, i.e. the length of the thread vector, is still below the maximum set for it.

Table 13. Axioms for the strategic interleaving operator $\|_{csi,f}(\_)$

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

Table 14. Axioms for the strategic interleaving operator $\|^{k,l}_{csi,f}(\_)$

$\|^{k}_{csi,f}(x) = \|^{k,l}_{csi,f}(x)$	CSIscf0
$\|^{k,l}_{csi,f}(\langle\rangle) = \mathsf{S}$	CSIscf1
$\|^{k,l}_{csi,f}(\langle\mathsf{S}\rangle \frown \alpha) = \|^{k,1}_{csi,f}(\alpha)$	CSIscf2
$\|^{k,l}_{csi,f}(\langle\mathsf{D}\rangle \frown \alpha) = \mathsf{S_D}(\|^{k,1}_{csi,f}(\alpha))$	CSIscf3
$\|^{k,l}_{csi,f}(\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha) =$	
$(\|^{k,1}_{csi,f}(\alpha \frown \langle x\rangle) \trianglelefteq a \trianglerighteq \|^{k,1}_{csi,f}(\alpha \frown \langle y\rangle))$	
$\triangleleft\, k = l \,\triangleright$	
$(\|^{k,l+1}_{csi,f}(\langle x\rangle \frown \alpha) \trianglelefteq a \trianglerighteq \|^{k,l+1}_{csi,f}(\langle y\rangle \frown \alpha))$	CSIscf4
$\|^{k,l}_{csi,f}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) =$	
$(\|^{k,1}_{csi,f}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|^{k,1}_{csi,f}(\alpha \frown \langle y\rangle))$	
$\triangleleft\, k = l \,\triangleright$	
$(\|^{k,l+1}_{csi,f}(\langle x\rangle \frown \alpha \frown \langle z\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|^{k,l+1}_{csi,f}(\langle y\rangle \frown \alpha))$	CSIscf5

3.6 Forking Combined with Step Counting

Features can be combined by integrating different equation systems for specifications of interleaving strategies. Table 14 displays the result of combining in a single scheduling operator both multi-threading with forks and step counting per thread (without yielding). In CSIscf4, a stands for an arbitrary action from $\mathsf{A_{tau}}$ 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 only proceed in one way: its processing never fails. For this strategy, axiom CSIf5 from Table 13 must be replaced by axiom CSIbf5 from Table 15. Here, an additional test action ?NT is used. Its processing succeeds if thread forking is enabled and its processing fails if thread forking is disabled. The enabledness condition may, for example, be that the number of active threads is less than a given maximum.

Table 15. Alternative axiom for $\|_{csi,f}(\_)$ with blocking fork actions

$\|_{csi,f}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) =$	
$\quad \|_{csi,f}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \trianglelefteq ?\mathsf{NT} \trianglerighteq \|_{csi,f}(\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)$	CSIbf5

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 is 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 a deadlock because thread forking may become enabled by external events at any time. Instead, the test action $?\mathsf{NT}$ is repeatly performed until it succeeds.

Another interpretation of blocking and failure is consistent with these equations as well: a thread forking is blocked if there is no free processor (assuming a multi-processor system), whereas it fails if the number of active threads is too high, the latter being viewed as a programming error. Here it should be assumed that the execution of a polarized process invokes a process with a bounded number of threads, while the number of processors allocated to that process may vary in time due to circumstances not under the control of the process or its execution engine. Moreover it is considered useful to admit at most one active thread per available processor. Thus thread forking requests for a processor, and in case that is currently unavailable, thread forking is blocked. This is reasonable as the process cannot have a bookkeeping of available processors. It may, however, setup its own bookkeeping of active threads and an attempt to have too many threads active at the same time may be considered a design flaw meriting a failure (exception) rather than blocking.

Leaving out the step counting for the moment this leads to a strategy with fork blocking and fork failure as in Table 16, which should be regarded as an extension of Table 5, after renaming the interleaving strategy operator $\|_{csi}(\_)$ to $\|_{csi,bff}(\_)$. This is in our opinion the most convincing description of forking that can be found in the current setting.

Step counting can easily be added to the previous strategy. It leads to a strategy as in Table 17, which should be regarded as an extension of Table 9, after renaming the interleaving strategy operator $\|^{k,l}_{csi}(\_)$ to $\|^{k,l}_{csi,bff}(\_)$.

Table 16. Axiom for $\|_{csi,bff}(\_)$, i.e. $\|_{csi}(\_)$ with blocking and failing fork actions

$\|_{csi,bff}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \curvearrowright \alpha) =$	
$\quad(\|_{csi,bff}(\alpha \curvearrowright \langle z\rangle \curvearrowright \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|_{csi,bff}(\alpha \curvearrowright \langle y\rangle))$	
$\quad\trianglelefteq ?\mathsf{NT} \trianglerighteq$	
$\quad\|_{csi,bff}\ (\alpha \curvearrowright \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)$	CSIbff5

Table 17. Axiom for $\|^{k,l}_{csi,bff}(\_)$, i.e. $\|^{k,l}_{csi}(\_)$ with blocking and failing fork actions

$\|^{k,l}_{csi,bff}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \curvearrowright \alpha) =$	
$\quad(\ (\|^{k,1}_{csi,bff}(\alpha \curvearrowright \langle z\rangle \curvearrowright \langle x\rangle) \triangleleft k = l \triangleright \|^{k,l+1}_{csi,bff}(\langle x\rangle \curvearrowright \alpha \curvearrowright \langle z\rangle))$	
$\quad\trianglelefteq \mathsf{NT} \trianglerighteq$	
$\quad(\|^{k,1}_{csi,bff}(\alpha \curvearrowright \langle y\rangle) \triangleleft k = l \triangleright \|^{k,l+1}_{csi,bff}(\langle y\rangle \curvearrowright \alpha))$	
$\quad)$	
$\quad\trianglelefteq ?\mathsf{NT} \trianglerighteq$	
$\quad\|^{k,1}_{csi,bff}\ (\alpha \curvearrowright \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)$	CSIscbff5

3.9 Terminating a Named Thread

Threads may influence one-another during their life-time. In this section it will be assumed that (i) threads are named by positive natural numbers which, (ii) should occur as the first parameter of the thread forking action $\mathsf{NT}(k, x)$, and (iii) forking is possible unless a thread with the intended name already exists. Threads present initially are all given name 0. 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: $\beta - k$ is the sequence obtained from β by removing k if it occurs in it and β itself otherwise. Secondly, $\rho_{\beta-k}(\alpha)$ removes from thread vector α the thread(s) named k if k occurs in β and leaves the thread vector unchanged otherwise. The new features then comprise the ability to 'terminate' a named thread from within another (or in fact the same) thread, and the option to test if a named thread is still alive. Equations for $\|^{\beta}_{csi,fn}(\_)$ are presented in Table 18 which is based on Table 13, ignoring the possibility that thread forking is blocking. In CSIfn4, a stands for an arbitrary action from $\mathsf{A}_{\mathsf{tau}}$ 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 $\mathsf{isalive?}k$.

4 Focus, Method and Guard

A useful format for basic actions is the so-called focus method notation (FMN) of [5]. In FMN an action consists of two parts called focus and method respectively. The focus comes first and is separated from the method with a dot. The

Table 18. Axioms for the strategic interleaving operator $\|^{\beta}_{csi,fn}(\_)$

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

focus indicates a service in an execution architecture for which the action represents a command, the method denotes that command proper. The syntax of foci and methods is the same: a nonempty sequence of alphanumeric strings separated by colons with the additional constraint that the first sequence is non-empty e.g '$f2{:}g{::}hh$' but not '$:ff{:}g$'. Complete actions in FMN are e.g: '$stack{:}a4.push{:}23$' or '$system{:}my.threadvector{:}currentthread{:}remove$'. In the absence of a focus some default focus is meant. In this paper the default focus is the thread vector TV. The question mark and exclamation mark of the manipulation instructions for threads terminate! and isalive? are simply turned into a semi-colon in FMN. Thus, in order to phrase these actions in terms of FMN one obtains: TV.terminate:k and TV.isalive:k.

Now we will extend FMN to FMN_{en} by introducing for each focus f and method m a basic action $f?m$, which issues the request to the service with focus f whether it is able to process method m. Basic actions of this form will be called enabledness requests, guarding tests or simply guards. If the reply is positive m is said to be enabled, otherwise it is blocked. It is an important constraint that processing an enabledness request cannot change the state of any service.

Internal steps (tau) are not connected to a particular focus. An internal step is always considered to be enabled. This is clear if one understands that internal steps arise from calculations on threads and services as the residue of the execution of enabled actions.

Threads are supposed initially not to contain any guarding tests for information on whether or not a method is enabled, because it is the task of the strategic interleaving to deal with blocking actions and enabledness of actions, by making use of guarding tests. In other words the SIOPs may (but need not) introduce guarding tests.

We will discuss a number of cyclic interleaving strategies in the absence of forking first to keep things simple and readable. The axioms for cyclic inter-

Table 19. Axioms for the strategic interleaving operator $\|_{csi,ba}(\_)$

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

leaving with blocking actions, denoted with $\|_{csi,ba}(\_)$, are given in Table 19.

The Role of a Focus When used in a thread, or more precisely stated in the description of a basic action that occurs in a thread a focus plays the role of a name of a service to which the action will be issued. The action proper is given by the method. Thus 'f:2.m:3' issues request 'm:3' to the service named 'f:2'. Different services may accept the same request and execute these differently. Below the notion of an execution architecture will be used for an environment that provides services for a set of foci. In an execution environment a focus provides a name for a service.

Classification of Foci A focus provides a handle to a service in an execution environment. These services are classified in either of 4 categories: target local services (not used in the sequel), para-target local services, target shared services and para-target shared services. It will be assumed that this classification can be derived from the foci, so that three sets of foci are used: F_{ptls} for all foci under which a para-target local service is known, F_{ptss} for all foci under which a para-target shared service is known, and F_{tss} for all foci under which a target shared service is known.

Test Result Preservation and Focus Classification For a focus f in F_{ptls} if a test $f?m$ has been performed and produces a negative reply this means that the thread on which behalf the test has been performed will deadlock. No other thread or external factor may influence the state of the service with focus f, and being the behavior of a sequential program the thread has no alternative than to proceed with $f?m$ which is a blocked action. There is no reason to repeat the test at a later stage as its outcome will be the same. Because of this fact it is reasonable to combine a new thread with the para-target local services of which the focus occurs in actions that are contained in the thread. This is formalized in detail in Section 7.3. For a focus in F_{ptls} it will be assumed that there is at most one thread that contains actions in which that focus occurs.

For a focus g in F_{tss} if a test $g?m$ has been performed with result b, the state of the service with focus g may change at any stage due to external causes. For that reason it is practical to repeat the test that had a negative reply after some time, thus waiting for a moment where the action $g.m$ is enabled. There is no reason to record any outcome of the test $g?m$ in order to be consulted by (the program implementing) some SIOP.

For a focus h in F_{ptss} if a test $h?m$ has been performed with result F, the state of the service with focus h may change whenever, and only if some other thread succeeds in executing an action of the form $h.m'$ for some method m'. For that reason it is only practical to repeat test after a negative reply after an event $h.m'$ has occurred. Each negative outcome of a test of the form $h?m$ should be recorded in order to be consulted by some SIOP. Recorded tests need not be repeated. But if an action is performed all negative results of tests of the form may be dropped ($V = V - h.$) as each test may have a different result again.

Avoiding Redundant Tests Strategic interleaving according to Table 19 has several obvious shortcomings: i) if all threads are blocked there is no loop detection in place, ii) the same test may be applied twice or more without any state change in between, iii) the first thread to request some action may not be the first thread granted permission to do so. The first two of these problems can be met by maintaining the set of all tests of the form $h?m$ with $h \in \mathrm{F}_{ptss}$ that have failed since the last action of the form $h.m'$. Such tests need not be repeated before they have been removed from this set. The axioms for this interleaving strategy with memory are given in Table 20. The auxiliary function $V - f.$ removes from V all actions with focus f. The notation $f.$ is used for the collection of all basic actions with focus f. The superscript i is a counter which counts the number of 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 service this counter is increased. If a target service is found blocked busy waiting takes place because at some later stage of the computation some external activity influencing the state of the (blocked) target service may allow the request to be granted even without any intermediate activity of the multi-thread under execution. In CSIbam6 this counter is reset to 0 after an action has been executed by a para-target shared service because it is not known how many threads have been blocked by actions for the same service and the 'worst case' must be taken into account. If all threads are blocked, i.e. the number of threads equals the number of threads that are certainly blocked, a deadlock is unavoidable.

5 Locking and Unlocking

A typical role for a service is that of a data type server, another role is that of an external device driver. A para-target service does nothing else but maintaining a state and producing replies on the basis of that state. Thus a para-target service

Table 20. Axioms for the strategic interleaving operator $\|^{V,i}_{csi,bam}(\_)$

$\|_{csi,bam}(\alpha) = \|^{\emptyset,0}_{csi,bam}(\alpha)$	CSIbam0
$\|^{V,i}_{csi,bam}(\langle\,\rangle) = \mathsf{S}$	CSIbam1
$\|^{V,i}_{csi,bam}(\langle\mathsf{S}\rangle \frown \alpha) = \|^{V,0}_{csi,bam}(\alpha)$	CSIbam2
$\|^{V,i}_{csi,bam}(\langle\mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\|^{V,i}_{csi,bam}(\alpha))$	CSIbam3
$\|^{V,i}_{csi,bam}(\langle\mathsf{tau} \circ x\rangle \frown \alpha) = \mathsf{tau} \circ \|^{V,i}_{csi,bam}(\alpha \frown \langle x\rangle)$	CSIbam4
$f \in \mathrm{F}_{ptls} \Rightarrow \|^{V,i}_{csi,bam}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) =$	
$\quad (\|^{V,i}_{csi,bam}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V,i}_{csi,bam}(\alpha \frown \langle y\rangle))$	
$\quad \trianglelefteq f?m \trianglerighteq$	
$\quad \|^{V,i}_{csi,bam}(\alpha \frown \langle\mathsf{D}\rangle)$	CSIbam5
$f \in \mathrm{F}_{ptss} \Rightarrow \|^{V,i}_{csi,bam}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) =$	
D	
$\triangleleft\, i \geq \mathsf{length}(\alpha) \triangleright$	
$(\ \|^{V,i+1}_{csi,bam}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	
$\quad \triangleleft\, f.m \in V \triangleright$	
$\quad (\ (\|^{V-f.,0}_{csi,bam}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V-f.,0}_{csi,bam}(\alpha \frown \langle y\rangle))$	
$\quad\quad \trianglelefteq f?m \trianglerighteq$	
$\quad\quad \|^{V\cup\{f.m\},i+1}_{csi,bam}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	
$\quad)$	
$)$	CSIbam6
$f \in \mathrm{F}_{tss} \Rightarrow \|^{V,i}_{csi,bam}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) =$	
$\quad (\|^{V,i}_{csi,bam}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V,i}_{csi,bam}(\alpha \frown \langle y\rangle))$	
$\quad \trianglelefteq f?m \trianglerighteq$	
$\quad \|^{V,i}_{csi,bam}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	CSIbam7

functions as a data type server. Technically a para-target service is a function $F : \Sigma^+ \to \{\mathsf{T}, \mathsf{F}\}$, where Σ is the set of methods that the service is able to process. This function is called the reply function of the para-target service. It determines the reply produced on the processing of a sequence of methods from Σ.

In the case where it is possible that certain methods get blocked, the reply function delivers D in case a method is not enabled. It is assumed, moreover that all enabledness requests are themselves always enabled, and produce correct results as well. Thus the reply function for a para-target service with blocking is a mapping $F : (\Sigma \cup ?\Sigma)^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{D}\}$ that satisfies the following constraints: $F(\sigma \frown \langle m\rangle) \in \{\mathsf{T}, \mathsf{F}\} \Leftrightarrow F(\sigma \frown \langle ?m\rangle) = \mathsf{T}$ and $F(\sigma \frown \langle m\rangle) = \mathsf{D} \Leftrightarrow F(\sigma \frown \langle ?m\rangle) = \mathsf{F}$ and $F(\sigma) = \mathsf{D} \Rightarrow F(\sigma \frown \langle ?m\rangle) = \mathsf{D}$.

Now a para-target shared service with locking is equipped with two methods lock and unlock. In all states either of the two methods is enabled, and its

execution moves the service to some state in which the other one is enabled. It is assumed that initially lock is enabled.

All threads are supposed to work accordingly to the concept of locking as follows. If a thread successfully performs the action $f.\mathsf{lock}$ it is said to acquire the lock (for f). Now it possesses the lock until the lock is released by performing $f.\mathsf{unlock}$. Now for each focus f in F_{ptss} all commands $f.m$ must be performed in a phase in which the thread is in possession of the lock of f. If threads adhere this rule (a matter to be ensured by the procedures for generating and or accepting threads by the system) and under the assumption that the para-target shared service works as stated by preventing successive lock-ings without intermediate unlock-ing, it is guaranteed that (i) at most a single thread can possess the lock for a para-target shared service at a give stage and (ii) a thread in possession of the lock for a para-target shared service has exclusive access to that service.

5.1 A Single Thread Deadlock

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

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

In P the second attempt to obtain the lock of f fails and the execution will deadlock. By requiring that threads alternate locking and unlocking of the same lock single thread deadlocks disappear just like in the case of Java which features no single thread deadlocks either.

5.2 Deadlocks and their 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 generated by target shared services. For instance, if $f \in \mathrm{F}_{tss}$, the thread vector $\langle \mathsf{S} \trianglelefteq f.m \trianglerighteq \mathsf{D}\rangle$ will deadlock under $\|_{csi,bam}(\_)$ after a negative reply by the service in focus f. After a positive reply it properly terminates, and in the case of blocking it waits till $f.m$ is found enabled.

More interesting deadlocks are those which arise from the coexistence of threads, and which would disappear in the absence of one or more of the threads. In particular deadlocks may be entirely due to the interaction with para-target shared services and para-target local services. 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 of an 'interesting' deadlock is as follows, where f and g are foci of different para-target shared services.

$$\begin{aligned} P = \|_{csi,bam}(&\langle f.\mathsf{lock} \circ g.\mathsf{lock} \circ g.\mathsf{unlock} \circ f.\mathsf{unlock} \circ \mathsf{S}\rangle \curvearrowright \\ &\langle g.\mathsf{lock} \circ f.\mathsf{lock} \circ f.\mathsf{unlock} \circ g.\mathsf{unlock} \circ \mathsf{S}\rangle). \end{aligned}$$

The 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. idle is like tau but it needs a focus, unlike tau. Now the following system deadlocks as well:

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

This deadlock works the same way as the previous one taking into account that the first thread makes one redundant step.

Placing both threads in a different order in the thread vector, however, makes the deadlock disappear, because now the first thread can take both locks before the second thread is able to perform a locking action:

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

Indeed in the case of R the step $h.\mathsf{idle}$ prevents the second thread from prematurely taking the lock that the first thread will need to complete its execution.

5.3 Deadlocks and their Dependence on Interleaving Strategy

Deadlock behavior is dependent on the interleaving strategy as well as on thread ordering. To substantiate that point we will first develop a step counting version of the strategy capable of dealing with method blocking. To simplify the matter para-target local services are not considered in this strategy. Table 21 contains equations for this strategy. Thread forking has been added as well. The underlying conceptual decision is that a deadlock can occur only if all threads are blocked and none of the threads attempts a fork. The idea is that $\|^1_{csi,bafm}(\_)$ coincides with $\|_{csi,bam}(\_)$ on all thread vectors (not involving forks of course). Now in connection with deadlock behavior the following can be observed:

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

For α one may simply take the thread vector used in P in Section 5.2. The following example for β combines previous ideas. In $\|^2_{csi,bafm}(\beta)$ in the first round both threads process the locking and subsequent unlocking of e. In the second round both threads take their first lock, and in the third round both threads are blocked. When executing $\|^1_{csi,bafm}(\beta)$ thread X takes the lock on e, then Y is blocked. Subsequently X releases the lock on e, which is then taken by Y. In the next round X takes the lock on f and Y releases the lock on e. Then both

Table 21. Axioms for the strategic interleaving operator $\|^{V,k,l,i}_{csi,bafm}(\_)$

$\|^{k}_{csi,bafm}(x) = \|^{\emptyset,k,0,0}_{csi,bafm}(x)$	CSIscbam0
$\|^{V,k,l,i}_{csi,bafm}(\langle\,\rangle) = \mathsf{S}$	CSIscbafm1
$\|^{V,k,l,i}_{csi,bafm}(\langle\mathsf{S}\rangle \frown \alpha) = \|^{V,k,0,0}_{csi,bafm}(\alpha)$	CSIscbafm2
$\|^{Vk,l,i}_{csi,bafm}(\langle\mathsf{D}\rangle \frown \alpha) = \mathsf{S}_{\mathsf{D}}(\|^{V,k,0,0}_{csi,bafm}(\alpha))$	CSIscbafm3
$\|^{Vk,l,i}_{csi,bafm}(\langle\mathsf{tau} \circ x\rangle \frown \alpha) = \mathsf{tau} \circ$	
$(\|^{Vk,l,1}_{csi,bafm}(\alpha \frown \langle x\rangle) \lhd k = l \rhd \|^{Vk,l+1}_{csi,bafm}(\langle x\rangle \frown \alpha))$	CSIscbafm4
$\|^{V,k,l,i}_{csi,bafm}(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) =$	
D	
$\lhd i \geq \mathsf{length}(\alpha) \rhd$	
$(\ \|^{V,k,1,i+1}_{csi,bafm}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	
$\lhd f.m \in V \rhd$	
$(\ (\ (\|^{V-f.,k,1,0}_{csi,bafm}(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \|^{V-f.,k,1,0}_{csi,bafm}(\alpha \frown \langle y\rangle)$	
$\lhd l = k \rhd$	
$(\|^{V-f.,k,l+1,0}_{csi,bafm}(\langle x\rangle \frown \alpha) \trianglelefteq f.m \trianglerighteq \|^{V-f.,k,l+1,0}_{csi,bafm}(\langle y\rangle \frown \alpha))$	
$)$	
$\trianglelefteq f?m \trianglerighteq$	
$(\ \|^{V,k,l,i}_{csi,bafm}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	
$\lhd f \notin \mathrm{F}_{ptss} \rhd$	
$\|^{V \cup \{f.m\},k,l,i+1}_{csi,bafm}(\alpha \frown \langle x \trianglelefteq f.m \trianglerighteq y\rangle)$	
$)$	
$)$	
$)$	CSIscbafm5
$\|^{V,k,l,i}_{csi,bafm}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) =$	
$(\ (\|^{V,k,1,i}_{csi,bafm}(\alpha \frown \langle z\rangle \frown \langle x\rangle) \lhd k = l \rhd \|^{V,k,l+1,i}_{csi,bafm}(\langle x\rangle \frown \alpha \frown \langle z\rangle))$	
$\trianglelefteq \mathsf{NT} \trianglerighteq$	
$(\|^{V,k,1,i}_{csi,bafm}(\alpha \frown \langle y\rangle) \lhd k = l \rhd \|^{V,k,l+1,i}_{csi,bafm}(\langle y\rangle \frown \alpha))$	
$)$	
$\trianglelefteq ?\mathsf{NT} \trianglerighteq$	
$\|^{V,k,1,i}_{csi,bafm}(\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)$	CSIscbafm6

threads perform a silent step, after which X can take its second lock. Because of the particular setup the strategy $\|^{1}_{csi,bafm}(\_)$ causes Y to be processed so much slower then in the two step cyclic strategy (in comparison to X of course), that X can acquire both locks simultaneously, which fails in the two step strategy.

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

$$Y = \langle e.\mathsf{lock} \circ e.\mathsf{unlock} \circ h.\mathsf{idle} \circ g.\mathsf{lock} \circ f.\mathsf{lock} \circ h.\mathsf{idle} \circ f.\mathsf{unlock} \circ h.\mathsf{idle} \circ g.\mathsf{unlock} \circ h.\mathsf{idle} \circ \mathsf{S}\rangle.$$

Formalized proofs of the properties of these examples, as well as the other examples from Sections 5.2 and 5.3, require a formalization of the phenomenon of a deadlock. This is postponed till Section 7.2, where a formalization is given in terms of an operator, called the use operator, introduced in Section 7.1.

Examples without an Explicit Thread Vector One might say that these examples concerning strategy dependence of deadlock behavior make use of an element foreign to common programming namely the explicit construction of a thread vector. Below the examples have been adapted to polarized processes with forks instead. The execution of forks gives rise to thread vectors, but no 'user control' of thread vectors is presupposed. Moreover it is taken for granted that forking always succeeds. The process P' generates after one execution step of a cyclic non-counting strategy the initial configuration of P from Section 5.2, and it will also deadlock, R' on the other hand will not deadlock.

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

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

5.4 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 PGA can be used. We refer to [5] for PGA. An extension with a fork instruction can be as follows: $\mathsf{fork}\#k; X$ starts up a forked thread which semantically coincides with $\#k; X$. At the same time it returns a boolean reply reporting on the success or failure of the thread creation. If true, the current thread proceeds as X, otherwise as $\#2; X$. Recalling from [5] that $|X|$ represents the extraction of a polarized process (thread) from a program, additional equations for process extraction in the case of a fork instruction read as follows:

$$|\mathsf{fork}\#\mathsf{k}; X| = \mathsf{NT}(|\#k; X|) \circ |X|$$
$$|{+}\mathsf{fork}\#\mathsf{k}; X| = |\#1; X| \trianglelefteq \mathsf{NT}(|\#k; X|) \trianglerighteq |\#2; X|$$
$$|{-}\mathsf{fork}\#\mathsf{k}; X| = |\#2; X| \trianglelefteq \mathsf{NT}(|\#k; X|) \trianglerighteq |\#1; X|$$

5.5 A Methodological Implication

A consequence of the observations made in Section 5.3 is that a manifestation of a deadlock for some thread vector on one system (modeled by some interleaving strategy), has no implications for the deadlock behavior of the same thread vector on another system (modeled by another 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 a 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 that he/she has in mind a general survey theory of strategic interleaving which permits a precise interpretation of the mentioned existential quantifier needed is unwarranted.

As a consequence the notion of a 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 a 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, however, 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 order to understand the working of a multi-threaded program on a machine in its simplest form the setup of execution architectures such as proposed in [7] is used and extended from the single thread case to the case involving a thread vector. Now 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 inter-

leaving operator, a polarized process is obtained, the process or multi-thread under execution, which operates in the context of the services.[3]

If the multi-thread under execution issues a command (basic action) $f.m$ and the architecture contains a service F with focus f, then control is given to F to process method m. Processing m may lead to a state change in F and it may also involve interaction of F with the environment of the architecture. In any case, after the processing is completed, control is returned to the process under execution, together with a mandatory boolean reply value. If the architecture does not contain a service with focus f, its use turns the thread issuing the command into D.

A para-target service is deterministic in its replies to the multi-thread under execution and it has no other interaction with anything inside or outside the execution architecture. It might equally be called a state machine component. Para-target services are used by a multi-thread under execution in the sense that commands for a para-target service are only issued in order to 'profit' from the boolean reply that is returned. A para-target service may be used as a shared memory for the threads in a multi-thread under execution.

After process execution has terminated (if ever) a para-target service will immediately be reset to its initial state. Output cannot be generated by a para-target service and persistent data cannot be stored by a para-target service. Recall that services that are not para-target services are called target services. Typically a target service may print some data, send an email, read some sensor data or activate other mechanical equipment. The overall intuition is that:

1. programs are written by people or machines for some purpose;
2. programs denote polarized processes, which can be interleaved according to various strategies;
3. given some interleaving strategy, realized by an execution architecture, the resulting polarized process is a multi-thread under execution, thereby inducing machine behavior, which if all goes well is intended behavior;
4. the intentions about that behavior only pertain to interactions with target services;
5. interactions with para-target services takes only place in as far as it is needed to generate the intended behavior in relation to target services.

7 Formalizing Para-Target Services

Para-target services as discussed in Section 1.3 extend the state machine behaviors of [6] by allowing commands to be blocked. Because a para-target service has an auxiliary role only and admits no interaction with any external system components its behavior may be formalized, following [6], by means of reply functions. The definition of reply functions below takes blocking into account.

[3] It operates exactly in the way a polarized processes operates in its execution environment as proposed in [7].

Table 22. Axioms for $\_ /_f \_$

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

In order to keep the notation as simple as possible enabledness test actions are left implicit in the definition of a reply function.

With K_{focus} we will denote the collection of foci and with K_{meth} we will denote the collection of methods. The same set of methods will be used for all services. A para-target service is formally given by a mapping, the so-called reply function, $F : {K_{meth}}^+ \rightarrow \{\mathsf{T}, \mathsf{F}, \mathsf{D}\}$, with the property that $F(\alpha) = \mathsf{D} \Rightarrow F(\alpha \frown \langle m \rangle) = \mathsf{D}$ for all sequences of methods α and for all methods m. It is a reasonable assumption that a reply function for a para-target service is computable.

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 $\frac{\partial}{\partial m} F(\alpha) = F(\langle m \rangle \frown \alpha)$.

7.1 Use Operator for Para-Target Services

The connection between a polarized process x and a para-target service F is given by the use operator, which expresses that in the architecture at hand F is present under focus f, and the commands along focus f of x are to be processed by F. This operator is justified because the para-target services are not modified or used by any other system component except the threads in the thread vector. The defining equations for the use operator are in Table 22.

Because the conditional operator $\_ \triangleleft \_ \triangleright \_$ is used with the additional truth value D, a third defining equation is needed for this operator: $x \triangleleft \mathsf{D} \triangleright y = \mathsf{D}$.

The use operator permits an encapsulation of the para-target services in the polarized process representing the thread vector after strategic interleaving.

Two use operator applications are independent if they concern different foci. In that case the order of application does not matter.

Theorem 7.1 (Commuting Uses).
$f \not\equiv g \Rightarrow (x /_f F) /_g G = (x /_g G) /_f F$

Proof. This has been demonstrated in a setting without blocking methods in [6]. 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: $(x /_f F) /_f G = x /_f F$.

7.2 The Phenomenon of a Deadlock

The use operator permits us to give a precise definition of deadlock behavior. This requires a notation for repeated internal steps:

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

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

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

Proposition 7.1 (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 deadlocks in Sections 5.1, 5.2 and 5.3.

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

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

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 is made of the following properties of the locking mechanism:

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

[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 deadlocks should involve thread vectors with D-free threads.

Table 23. Additional axiom for $\|_{csi,bff}(\_)$ with initialization of a local service

$\|_{csi,bff}(\langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle \frown \alpha) =$	
$(\|_{csi,bff}(\alpha \frown \langle z /_f F_{init}\rangle \frown \langle x\rangle) \trianglelefteq \mathsf{NT} \trianglerighteq \|_{csi,bff}(\alpha \frown \langle y\rangle))$	
$\trianglelefteq ?\mathsf{NT} \trianglerighteq$	
$\|_{csi,bff}\ (\alpha \frown \langle x \trianglelefteq \mathsf{NT}(z) \trianglerighteq y\rangle)$	CSIbffps5

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

7.3 Initialized Para-Target Local Services

Suppose that each thread is granted the use of para-target local service F, with focus f, with initial state F_{init}. With help of the use operator a strategy can be specified in this case. If the matter is considered without step counting and test memory, it amounts to the replacement of the equation for thread creation from Table 16 by the equation given in Table 23.[5]

A weakness of the setup in Table 23 lies in the fact 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 Static Methods

In object oriented program notations recursion in the form of recursive method calls plays an essential role. In this Section an extension of the thread algebra is given which treats classes with static methods at the level of threads. This is needed for modeling synchronization in Java for instance. Although synchronization is just a matter of locking and unlocking, several particular phenomena take place. If a thread has claimed (i.e. locked) a lock any subsequent attempt made by the thread to acquire the lock again (before releasing the lock) will succeed. In contrast, in the example in Section 5.1 a deadlock occurs because a thread attempts to acquire the same lock twice without unlocking in between. Moreover, Java provides means for a thread to temporarily release a lock, to

[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 the concept of a deadlock is apparent and where at the same time the task to define deadlocks in an entirely strategy independent fashion is not an obvious one.

Table 24. 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

hand it over to another thread and to claim it back in a later stage. This is the so-called 'wait' and 'notify' mechanism, which is connected with recursion as well.

Recursion, locking and unlocking in the presence of recursion, and the temporary release of a lock will be explained below in a way consistent with each of the previously discussed SIOPs that can deal with blocking actions.

In order to deal with recursion thread algebra is extended with two additional constants: $\mathsf{S}+$ and $\mathsf{S}-$ representing termination with a positive or a negative result. These are the polarized termination constants. Postconditional composition may now be extended by allowing the middle argument to be an entire thread. Its axioms appear in Table 24. Using polarized termination threads may appear as subroutines of other threads. With BPPApt the collection of polarized processes that may end in $\mathsf{S}+$, $\mathsf{S}-$ or in D is denoted.

8.1 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[6] name m and a thread p, where p is supposed always to end in a polarized termination or in D. A class description then has the form:

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

It will be said that 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

[6] Class methods are often referred to as static methods. Class methods are called with a class ($C..m$, with C a class and m a method), this in contrast to instance methods which are called with a reference to an object. Instance methods are not covered in this paper, however.

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 static 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 no definition of a method m is available in the environment, 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 static type checking techniques. We will use combined definitions of zero or more classes and processes that may involve class method calls on these classes. As a first example consider the definition

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

Then the semantics of class method calls implies:

$$P = g.m2 \circ (\mathsf{D} \trianglelefteq g.m1 \trianglerighteq (h.m3 \circ \mathsf{S})).$$

This illustrates that class method calls may be considered a generalization of service method calls. As a second example consider the following process definition:

$$\begin{aligned}
&class{:}Ca = \{\\
&\quad m1 = f.u1 \circ f.u2 \circ Ca..m2 \circ \mathsf{S}+,\\
&\quad m2 = \mathsf{S}+ \trianglelefteq g.v2 \trianglerighteq \mathsf{S}-,\\
&\quad m3 = f.u3 \circ f.u4 \circ \mathsf{S}+\\
&\quad \}\\
&class{:}Cb = \{\\
&\quad n1 = h.w1 \circ (\mathsf{S}+ \trianglelefteq h.w3 \trianglerighteq \mathsf{S}-),\\
&\quad m1 = \mathsf{S}- \trianglelefteq Ca.m2 \trianglerighteq \mathsf{S}+\\
&\quad \}\\
&P = h.w0 \circ (\mathsf{D} \trianglelefteq Cb..n1 \trianglerighteq (Ca..m3 \circ \mathsf{S}))
\end{aligned}$$

In this case one may derive:

$$
\begin{aligned}
&P = \\
&h.w0 \circ (\mathsf{D} \trianglelefteq h.w1 \circ (\mathsf{S}+ \trianglelefteq h.w3 \trianglerighteq \mathsf{S}-) \trianglerighteq (Ca..m3 \circ \mathsf{S})) = \\
&h.w0 \circ h.w1 \circ (\mathsf{D} \trianglelefteq h.w3 \trianglerighteq (Ca..m3 \circ \mathsf{S})) = \\
&h.w0 \circ h.w1 \circ (\mathsf{D} \trianglelefteq h.w3 \trianglerighteq (f.u3 \circ f.u4 \circ \mathsf{S}))
\end{aligned}
$$

8.2 Synchronized Methods

The class definition syntax is now adapted to allow an optional keyword 'synchronized' to precede the definition of a class method. It will be enclosed in brackets. To understand the purpose of this feature it is assumed that for each class C there is a unique service $classlock{:}C$ which will provide a lock for that class. If a synchronized class method is called, the class lock of the relevant class must be acquired first (by the thread calling the class method) and at the end of the execution of the body of the method the class lock is released again. This leads to an extended class and process definition syntax. The following example, features a thread vector rather than a single thread. Indeed synchronized methods are only relevant in the context of multi-threading.

$$
\begin{aligned}
&class{:}Ca = \{ \\
&\quad m1 = f.u1 \circ f.u2 \circ Ca..m2 \circ \mathsf{S}+, \\
&\quad (synchronized)\, m2 = \mathsf{S}+ \trianglelefteq g.v2 \trianglerighteq \mathsf{S}-, \\
&\quad m3 = f.u3 \circ f.u4 \circ \mathsf{S}+ \\
&\quad \} \\
&class{:}Cb = \{ \\
&\quad (synchronized)\, n1 = h.w1 \circ (\mathsf{S}+ \trianglelefteq h.w3 \trianglerighteq \mathsf{S}-), \\
&\quad m1 = \mathsf{S}- \trianglelefteq Ca.m2 \trianglerighteq \mathsf{S}+ \\
&\quad \} \\
&P = \langle h.w0 \circ (\mathsf{D} \trianglelefteq Cb..n1 \trianglerighteq (Ca..m3 \circ \mathsf{S}))\rangle \curvearrowright \langle Ca..m2 \circ \mathsf{S}\rangle
\end{aligned}
$$

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

$$
\begin{aligned}
&x \trianglelefteq C..m_i \trianglerighteq y = \\
&\quad (classlock{:}C.unlock \circ x) \\
&\qquad \trianglelefteq (classlock{:}C.lock \circ p_i) \trianglerighteq \\
&\quad (classlock{:}C.unlock \circ y)
\end{aligned}
$$

8.3 Lock Administration Operators for each Lock

Unfortunately the semantic description of synchronized class methods s given above falls short of giving an account of synchronized class methods in Java

Table 25. 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
$\mathsf{LA}^{C,n+1}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{lock} \trianglerighteq y) = \mathsf{tau} \circ \mathsf{LA}^{C,n+2}(x)$	LA5a
$\mathsf{LA}^{C,0}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{unlock} \trianglerighteq y) = \mathsf{D}$	LA5b
$\mathsf{LA}^{C,1}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{unlock} \trianglerighteq y) = \mathit{classlock}{:}C.\mathit{unlock} \circ \mathsf{LA}^{C,0}(x)$	LA6a
$\mathsf{LA}^{C,n+2}(x \trianglelefteq \mathit{classlock}{:}C.\mathit{unlock} \trianglerighteq y) = \mathsf{tau} \circ \mathsf{LA}^{C,n+1}(x)$	LA6b

because the same lock may be claimed twice in the case of recursion. Still the locking and unlocking actions have some importance even if a lock is claimed because only after as many unlockings as lockings the lock is in fact released. To take this into account threads are transformed by the lock administration operator ($\mathsf{LA}^{\text{-}}(\_)$) before being put in the thread vector. For each class used in a thread a different lock administration operator is required. The lock administration operator carries a count for each lock concerning the number of times it has been claimed thus making sure that the class lock is locked only once and subsequent claims on the lock as well as releases of it are granted automatically until the last one which requires unlocking the class lock. Lock administration operators are specified in Table 25. The use of the lock administration operators is exemplified by an adaptation of the example of Section 8.2 where the thread vector P is now given by:

$$
\begin{aligned}
&\mathit{class}{:}Ca = \{...\\
&\}\\
&\mathit{class}{:}Cb = \{...\\
&\}\\
&P =\\
&\quad \langle \mathsf{LA}^{Ca}(\mathsf{LA}^{Cb}(h.w0 \circ (\mathsf{D} \trianglelefteq Cb..n1 \trianglerighteq (Ca..m3 \circ \mathsf{S})))) \rangle \curvearrowright\\
&\quad \langle \mathsf{LA}^{Ca}(\mathsf{LA}^{Cb}(Ca..m2 \circ \mathsf{S})) \rangle
\end{aligned}
$$

8.4 Wait and Notify

Java's synchronization primitives involve actions 'wait' and 'notify' that can be applied to so-called object locks. A version of 'wait' that works for class locks (in the absence of 'notify') may be written as $C..\mathit{wait}$ with the defining equation:

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

Here 'wait' and 'unwait' are methods having exactly the same effect on the lock as 'lock' and 'unlock' 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 for '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. It is assumed that the lock for a class represents some condition 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 of acquiring the lock. Each thread blocked by that lock is waiting for the same condition ϕ^C to be satisfied. A typical example of such a condition is that some natural number m takes a positive (i.e. non-zero) value. Now an action '*C..notify*' represents a thread giving the message to 'the system' that it has now ensured that the condition ϕ^C holds true. Having done so, it may release the lock of C by either exiting its synchronized method body that claimed the lock or by performing an action '*C..wait*'. After the class lock has been released in this particular way (i.e. by a thread claiming the lock after a 'wait' performed by another thread owning the lock after providing a notification in advance) the interleaving strategy must ensure that one of the waiting threads (if any of those still exist) is permitted to acquire the lock, thus making sure that the condition ϕ^C holds when the lock is obtained by the waiting thread. A thread performing an action 'notify' need not immediately terminate 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 termination takes place.

In order to model notification the classlock service will be redesigned to a more involved service with an infinite state space. In addition to the existing states *classlock*:$C(locked)$ and *classlock*:$C(unlocked)$, for each $n > 0$ there are states

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

The notation is simplified by writing *classlock*:$C(locked) =$ *classlock*:$C(locked, 0)$ and *classlock*:$C(unlocked) =$ *classlock*:$C(unlocked, 0)$.

'*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*:$C(locked, 0)$ and
'*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)* and
'*wait*' bringing it into state *classlock*:$C(unlocked, n+1)$.
The method '*wait*' takes place if a thread in possession of the classlock for C performs '*classlock*:*C.wait*' thus becoming an additional waiting thread. The classlock 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, and
'*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 only the method
'*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 defined interleaving strategies provided these can deal with blocked methods. It should be used in combination with recursion and lock administration operators.

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

9 Different Processes on the Same Machines

A process in the sense of conventional operating system terminology may be identified with a multi-thread under execution. In order to prevent confusion

with the terminology from process theory as 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.

Deadlocks are 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. Multi-process execution models cannot be formalized as easily as multi-threads, however, 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, however minimal, including at least: starting a process from data containing a description of a startup thread (e.g. a program for it), terminating the process by a forced quit, monitoring termination and deadlocks and providing options for proceeding thereafter.

If all of these matters are ignored it is reasonable to model the cooperation of several system processes as the first strategic interleaving ($\|_{csi}(\_)$ as given in Table 5) 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 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.

10 Conclusions

We have outlined a theory of threads and multi-threads based on strategic interleaving. It has been developed to the amount of detail needed to capture a reasonable definition of deadlock. Then it is demonstrated that the occurrence of deadlocks thus defined depends on strategies and on the ordering of thread vectors.

The claim is put forward that the basic intuitions concerning deadlocks 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 deadlocks and deadlock freedom robust against a very wide choice of interleaving strategies is not considered a part of those intuitions. Thus, whereas we have succeeded in the specification

of a number of plausible interleaving strategies that demonstrate various key features concerning the cooperation of threads, at the same time 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, but we contend that in both approaches one will engage in a course of thought not immediately helpful, or needed for a programmer, whereas the theory outlined in this paper seems to convey no information that might be considered redundant or even counterproductive for a programmer who wants to develop 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 its executions.

Several topics for future research can be formulated along this line of work: the extension of this theory to a theory for parallel and distributed execution architectures; manageable models for concepts such as thread mobility and threads logging in to another machine in a network.

References

1. L. Aceto, W. J. Fokkink, and C. Verhoef. Structural operational semantics. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. Elsevier, Amsterdam, 2001.
2. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
3. J. A. Bergstra and I. Bethke. Polarized process algebra and program equivalence. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proceedings of ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2003.
4. J. A. Bergstra and J.-W. Klop. Process algebra for synchronous communication. *Information and Control*, 60 (1/3):109–137, 1984.
5. J. A. Bergstra and M. E. Loots. Program algebra for sequential code. *Journal of Logic and Algebraic Programming*, 51(2):125–156, 2002.
6. J. A. Bergstra and A. Ponse. Combining programs and state machines. *Journal of Logic and Algebraic Programming*, 51(2):175–192, 2002.
7. J. A. Bergstra and A. Ponse. Execution architectures for program algebra. Logic Group Preprint Series 230, Department of Philosophy, Utrecht University, Utrecht, 2004.
8. J. Bishop and N. Horspool. *C# Concisely*. Addison-Wesley, 2004.
9. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(8):560–599, 1984.
10. C. Jesshope. Implementing an efficient vector instruction set in a chip multiprocessor using micro-threaded pipelines. *Australian Computer Science Communications*, 23(4):80–88, 2001.
11. C. A. Middelburg. An alternative formulation of operational conservativity with binding terms. *Journal of Logic and Algebraic Programming*, 55(1/2):1–19, 2003.
12. R. Milner. *A calculus of communicating systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

13. M. R. Mousavi, M. A. Reniers, and J. F. Groote. Congruence for SOS with data. Computer Science Report 04-05, Department of Mathematics and Computer Science, Eindhoven University of Technology, January 2004.