A Thread Algebra with Multi-Level 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. In a previous paper, we developed an algebraic theory of threads and multi-threads based on strategic interleaving. This theory includes a number of plausible interleaving strategies on thread vectors. The strategic interleaving of a thread vector constitutes a multi-thread. Several multi-threads may exist concurrently on a single host in a network, several host behaviors may exist concurrently in a single network on the internet, etc. Strategic interleaving is also present at these other levels. In the current paper, we extend the theory developed so far with features to cover multi-level strategic interleaving.

1 Introduction

A thread is the behavior of a deterministic sequential program under execution. Multi-threading refers to the concurrent existence of several threads in a program under execution. Multi-threading is the dominant form of concurrency provided by recent object-oriented programming languages such as Java [1] and C# [2]. Arbitrary interleaving, on which theories about concurrent processes such as ACP [3] are based, is not the appropriate intuition when dealing with multi-threading. In the case of multi-threading, some deterministic interleaving strategy is used. In [4], we introduced a number of plausible deterministic interleaving strategies for multi-threading. We also proposed to use the phrase strategic interleaving for the more constrained form of interleaving obtained by using such a strategy.

The strategic interleaving of a thread vector constitutes a multi-thread. In conventional operating system jargon, a multi-thread is called a process. Several multi-threads may exist concurrently on the same machine. Multi-processing refers to the concurrent existence of several multi-threads on a machine. Such machines may be hosts in a network, and several host behaviors may exist concurrently in the same network. And so on and so forth. Strategic interleaving

is also present at these other levels. In the current paper, we extend the theory developed so far with features to cover multi-level strategic interleaving. There is a dependence on the interleaving strategy considered. We extend the theory only for the simplest case: cyclic interleaving. Other plausible interleaving strategies are treated in [4]. They can also be adapted to the setting of multi-level strategic interleaving.

Threads proceed by performing steps, in the sequel called basic actions, in a sequential fashion. Performing a basic action is taken as making a request to a certain service provided by the execution environment to process a certain command. The service produces a reply value which is returned to the thread concerned. A service may be local to a single thread, local to a multi-thread, local to a host, or local to a network. In this paper, we introduce thread-service composition in order to bind certain basic actions of a thread to certain services.

An axiomatic description of multi-level strategic interleaving and thread-service composition, as well as a structural operational semantics, is provided. One of our objectives is to develop a simplified, formal representation schema of the design of systems that consist of several multi-threaded programs on various hosts in different networks. We propose to use the term formal design prototype for such a schema. Evidence of the correctness of the presented schema is obtained by a simulation lemma, which states that a finite thread consisting of basic actions that will not be processed by any available service is simulated by any instance of the schema that contains the thread in one of its thread vectors.

Thread algebra with multi-level strategic interleaving is a design on top of BPPA (Basic Polarized Process Algebra) [5, 6]. BPPA is far less general than ACP-style process algebras and its design focuses on the semantics of deterministic sequential programs. The semantics of a deterministic sequential program is supposed to be a polarized process. Polarization is understood along the axis of the client-server dichotomy. Basic actions in a polarized process are either requests expecting a reply or service offerings promising a reply. Thread algebra may be viewed as client-side polarized process algebra because all threads are viewed as clients generating requests for services provided by their environment.

The structure of this paper is as follows. After a review of BPPA, we extend it to a basic thread algebra with cyclic interleaving, but without any feature for multi-level strategic interleaving. Next, we extend this basic thread algebra with thread-service composition and other features for multi-level strategic interleaving. Following this, we discuss how two additional features can be expressed and give a formal representation schema of the design of systems that consist of several multi-threaded programs on various hosts in different networks. Finally, we make some concluding remarks.

2 Basic Polarized Process Algebra

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

Table 1. Axiom of BPPA

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

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

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

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

The intuition is that each basic action is taken as a command to be processed by the execution environment. The processing of a command may involve a change of state of the execution environment. At completion of the processing of the command, the execution environment produces a reply value. This reply is either T or F and is returned to the polarized process concerned. Let p and q be closed terms of BPPA. Then $p \trianglelefteq a \trianglerighteq q$ will proceed as p if the processing of a leads to the reply T (called a positive reply), and it will proceed as q if the processing of a leads to the reply F (called a negative reply). If the reply is used to indicate whether the processing was successful, a useful convention is to indicate successful processing by the reply T and unsuccessful processing by the reply F. The action tau plays a special role. Its execution will never change any state and always produces a positive reply.

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

Following [6], 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 the requirement that repeated substitution of the right-hand sides of equations for the left-hand side variables eventually produces an expression of the form D, S or $p \trianglelefteq a \trianglerighteq q$. For each guarded system of recursion equations E and each variable X that occurs as the left-hand side of an equation in E, we add to the constants of BPPA a constant standing for the unique solution of E for X. This constant is denoted by X_E.

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 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], which in fact uses the notation of [3], approximation of depth n is phrased in terms of a unary *projection* operator $\pi_n(\_)$. The projection operators are defined inductively by means of the axioms in Table 2. In this table and all subsequent tables with axioms in which

Table 2. Axioms for projection

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

a occurs, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$.

As mentioned above, the behavior of a polarized process depends upon its execution environment. Each basic action performed by the polarized process is taken as a command to be processed by the execution environment. At any stage, the commands that the execution environment can accept depend only on its history, i.e. the sequence of commands processed before and the sequence of replies produced for those commands. When the execution environment accepts a command, it will produce a positive reply 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.

In the structural operational semantics, we represent an execution environment by a function $\rho : (\mathcal{A} \times \{\mathsf{T}, \mathsf{F}\})^* \to \mathcal{P}(\mathcal{A} \times \{\mathsf{T}, \mathsf{F}\})$ that satisfies the following condition: $(a, b) \notin \rho(\alpha) \Rightarrow \rho(\alpha \frown \langle (a, b) \rangle) = \emptyset$ for all $a \in \mathcal{A}$, $b \in \{\mathsf{T}, \mathsf{F}\}$ and $\alpha \in (\mathcal{A} \times \{\mathsf{T}, \mathsf{F}\})^*$.[4] We write $\mathcal{E}$ for the set of all those functions. Given an execution environment $\rho \in \mathcal{E}$ and a basic action $a \in \mathcal{A}$, 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 $a \in \mathcal{A}_{\mathsf{tau}}$ 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 $\rho \in \mathcal{E}$.

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

- $\langle p, \rho \rangle \xrightarrow{a} \langle p', \rho' \rangle$: in execution environment ρ, process p is capable of first performing action a and then proceeding as process p' in execution environment ρ';
- $\langle p, \rho \rangle \downarrow$: in execution environment ρ, process p is capable of terminating successfully;

[4] We write $\langle\,\rangle$ for the empty sequence, $\langle d \rangle$ for the sequence having d as sole element, and $\alpha \frown \beta$ for the concatenation of sequences α and β. We assume that the identities $\alpha \frown \langle\,\rangle = \langle\,\rangle \frown \alpha = \alpha$ hold.

Table 3. Transition rules for BPPA with projection and recursion

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

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

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

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

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

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

The structural operational semantics of BPPA extended with projection and recursion is described by the transition rules given in Table 3. In this table and all subsequent tables with transition rules in which a occurs, a stands for an arbitrary action from $\mathcal{A}_{\mathsf{tau}}$. We write t_E 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 X_E.

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

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

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

Bisimulation equivalence is a congruence with respect to the postconditional composition operators and the projection operators. This follows immediately from the fact that the transition rules for BPPA with projection and recursion constitute a transition system specification in path format (see e.g. [7]).

3 Basic Thread Algebra with Foci and Methods

In this section, we introduce a thread algebra without features for multi-level strategic interleaving. Such features will be added in subsequent sections.

In [5], its has been outlined how and why polarized processes are a natural candidate for the specification of the semantics of deterministic sequential programs. Assuming that a thread is a process representing a deterministic se-

Table 4. Axioms for cyclic interleaving

$\parallel(\langle\rangle) = \mathsf{S}$	CSI1
$\parallel(\langle \mathsf{S}\rangle \frown \alpha) = \parallel(\alpha)$	CSI2
$\parallel(\langle \mathsf{D}\rangle \frown \alpha) = \mathsf{S}_\mathsf{D}(\parallel(\alpha))$	CSI3
$\parallel(\langle \mathsf{tau} \circ x\rangle \frown \alpha) = \mathsf{tau} \circ \parallel(\alpha \frown \langle x\rangle)$	CSI4
$\parallel(\langle x \trianglelefteq f.m \trianglerighteq y\rangle \frown \alpha) = \parallel(\alpha \frown \langle x\rangle) \trianglelefteq f.m \trianglerighteq \parallel(\alpha \frown \langle y\rangle)$	CSI5

Table 5. Axioms for deadlock at termination

$\mathsf{S}_\mathsf{D}(\mathsf{S}) = \mathsf{D}$	S2D1
$\mathsf{S}_\mathsf{D}(\mathsf{D}) = \mathsf{D}$	S2D2
$\mathsf{S}_\mathsf{D}(\mathsf{tau} \circ x) = \mathsf{tau} \circ \mathsf{S}_\mathsf{D}(x)$	S2D3
$\mathsf{S}_\mathsf{D}(x \trianglelefteq f.m \trianglerighteq y) = \mathsf{S}_\mathsf{D}(x) \trianglelefteq f.m \trianglerighteq \mathsf{S}_\mathsf{D}(y)$	S2D4

quential program under execution, it is reasonable to view all polarized processes as threads. A thread vector is a sequence of threads.

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. In this paper, we only cover the simplest interleaving strategy, namely *cyclic interleaving*. Other plausible interleaving strategies are treated in [4]. They can also be adapted to the features for multi-level level strategic interleaving that will be introduced in the current paper. The strategic interleaving operator for cyclic interleaving is denoted by $\parallel(\_)$. In [4], it was denoted by $\parallel_{csi}(\_)$ to distinguish it from other strategic interleaving operators.

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

The axioms for cyclic interleaving are given in Table 4. In this table and all subsequent tables with axioms or transition rules in which f and m occur, f and m stand for an arbitrary focus from $\mathcal{F}$ and an arbitrary method from $\mathcal{M}$, respectively. In CSI3, the auxiliary *deadlock at termination* operator $\mathsf{S}_\mathsf{D}(\_)$ is used. This operator turns termination into deadlock. Its axioms appear in Table 5.

The structural operational semantics of the basic thread algebra with foci and methods is described by the transition rules given in Tables 3 and 6. Here $\langle x, \rho\rangle \not\to$ stands for the set of all negative conditions $\neg(\langle x, \rho\rangle \xrightarrow{a} \langle p', \rho'\rangle)$ where p' is a closed term of BPPA, $\rho' \in \mathcal{E}$, $a \in \mathcal{A}_{\mathsf{tau}}$. Recall that $\mathcal{A} = \{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.

Bisimulation equivalence is also a congruence with respect to the cyclic interleaving operator and the deadlock at termination operator. This follows im-

Table 6. Transition rules for cyclic interleaving and deadlock at termination

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

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

$$\frac{\langle x_1,\rho\rangle\downarrow,\ldots,\langle x_k,\rho\rangle\downarrow}{\langle \| (\langle x_1\rangle \frown \ldots \frown \langle x_k\rangle),\rho\rangle\downarrow} \qquad \frac{\langle x_1,\rho\rangle \not\rightarrow,\ldots,\langle x_k,\rho\rangle \not\rightarrow,\langle x_l,\rho\rangle\uparrow}{\langle \| (\langle x_1\rangle \frown \ldots \frown \langle x_k\rangle),\rho\rangle\uparrow} \quad (k \geq l > 0)$$

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

mediately from the fact that the transition rules for the basic thread algebra with foci and methods constitute a complete transition system specification in relaxed panth format (see e.g. [8]).

4 Thread-Service Composition

In this section, we extend the basic thread algebra with foci and methods with thread-service composition. For each $f \in \mathcal{F}$, we introduce a *thread-service composition* operator $\_ \mathbin{/_f} \_$. These operators have a thread as first argument and a service as second argument. $P \mathbin{/_f} H$ is the thread that results from issuing all basic actions from thread P that are of the form $f.m$ to service H.

A service is represented by a function $H : \mathcal{M}^+ \to \{\mathsf{T},\mathsf{F},\mathsf{B},\mathsf{R}\}$ with the property that $H(\alpha) = \mathsf{B} \Rightarrow H(\alpha \frown \langle m\rangle) = \mathsf{B}$ and $H(\alpha) = \mathsf{R} \Rightarrow H(\alpha \frown \langle m\rangle) = \mathsf{R}$ for all $\alpha \in \mathcal{M}^+$ and $m \in \mathcal{M}$. This function is called the *reply* function of the service. Given a reply function H and a method m, the derived reply function of H after processing m, written $\frac{\partial}{\partial m}H$, is defined by $\frac{\partial}{\partial m}H(\alpha) = H(\langle m\rangle \frown \alpha)$.

The connection between a reply function H and the service represented by it can be understood as follows:

- If $H(\langle m\rangle) = \mathsf{T}$, the request to process command m is accepted by the service, the reply is positive and the service proceeds as $\frac{\partial}{\partial m}H$.
- If $H(\langle m\rangle) = \mathsf{F}$, the request to process command m is accepted by the service, the reply is negative and the service proceeds as $\frac{\partial}{\partial m}H$.
- If $H(\langle m\rangle) = \mathsf{B}$, the request to process command m is not refused by the service, but the processing of m is temporarily blocked. The request will have to wait until the processing of m is not blocked any longer.
- If $H(\langle m\rangle) = \mathsf{R}$, the request to process command m is refused by the service.

The axioms for thread-service composition are given in Table 7. In this table and all subsequent tables with axioms or transition rules in which g occurs, like f, g stands for an arbitrary focus from $\mathcal{F}$.

The structural operational semantics of the basic thread algebra with foci and methods extended with thread-service composition is described by the transition rules given in Tables 3, 6 and 8.

Table 7. Axioms for thread-service composition

$\mathsf{S} \mathbin{/_f} H = \mathsf{S}$		TSC1
$\mathsf{D} \mathbin{/_f} H = \mathsf{D}$		TSC2
$(\mathsf{tau} \circ x) \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} H)$		TSC3
$(x \trianglelefteq g.m \trianglerighteq y) \mathbin{/_f} H = (x \mathbin{/_f} H) \trianglelefteq g.m \trianglerighteq (y \mathbin{/_f} H)$	if $f \neq g$	TSC4
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ (x \mathbin{/_f} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{T}$	TSC5
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{tau} \circ (y \mathbin{/_f} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{F}$	TSC6
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_f} H = \mathsf{D}$	if $H(\langle m \rangle) \in \{\mathsf{B}, \mathsf{R}\}$	TSC7

Table 8. Transition rules for thread-service composition

$$\frac{\langle x, \rho\rangle \xrightarrow{g.m} \langle x', \rho'\rangle}{\langle x \mathbin{/_f} H, \rho\rangle \xrightarrow{g.m} \langle x' \mathbin{/_f} H, \rho'\rangle}\ f \neq g \qquad \frac{\langle x, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x', \rho'\rangle}{\langle x \mathbin{/_f} H, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x' \mathbin{/_f} H, \rho'\rangle}$$

$$\frac{\langle x, \rho\rangle \xrightarrow{f.m} \langle x', \rho'\rangle}{\langle x \mathbin{/_f} H, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x' \mathbin{/_f} \frac{\partial}{\partial m} H, \rho'\rangle}\ H(\langle m \rangle) \in \{\mathsf{T}, \mathsf{F}\},\ (f.m, H(\langle m \rangle)) \in \rho(\langle\,\rangle)$$

$$\frac{\langle x, \rho\rangle \xrightarrow{f.m} \langle x', \rho'\rangle}{\langle x \mathbin{/_f} H, \rho\rangle\!\uparrow}\ H(\langle m \rangle) \in \{\mathsf{B}, \mathsf{R}\} \qquad \frac{\langle x, \rho\rangle\!\downarrow}{\langle x \mathbin{/_f} H, \rho\rangle\!\downarrow} \qquad \frac{\langle x, \rho\rangle\!\uparrow}{\langle x \mathbin{/_f} H, \rho\rangle\!\uparrow}$$

The action tau arises as the residue of processing commands. Therefore, tau is not connected to a particular focus, and is always accepted.

5 Guarding Tests

In this section, we extend the thread algebra developed so far with guarding tests. Guarding tests are basic actions meant to verify whether a service will accept the request to process a certain method now, and if not so whether it will be accepted after some time. Guarding tests allow for dealing with delayed processing and exception handling as will be shown in Section 6.

We extend the set of basic actions. For the set of basic actions, we now take the set $\{f.m, f?m, f??m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Basic actions of the forms $f?m$ and $f??m$ will be called *guarding tests*. Performing a basic action $f?m$ is taken as making the request to the service named f to reply whether it will accept the request to process method m now. The reply is positive if the service will accept that request now, and otherwise it is negative. Performing a basic action $f??m$ is taken as making the request to the service named f to reply whether it will accept the request to process method m now or after some time. The reply is positive if the service will accept that request now or after some time, and otherwise it is negative.

As explained below, it happens that not only thread-service composition but also cyclic interleaving has to be adapted to the presence of guarding tests.

The additional axioms for cyclic interleaving and deadlock at termination in the presence of guarding tests are given in Table 9. Axioms CSI6 and CSI7

Table 9. Additional axioms for cyclic interleaving & deadlock at termination

$\parallel (\langle x \trianglelefteq f?m \trianglerighteq y\rangle \frown \alpha) = \parallel (\langle x\rangle \frown \alpha) \trianglelefteq f?m \trianglerighteq \parallel (\alpha \frown \langle y\rangle)$	CSI6
$\parallel (\langle x \trianglelefteq f??m \trianglerighteq y\rangle \frown \alpha) = \parallel (\langle x\rangle \frown \alpha) \trianglelefteq f??m \trianglerighteq \parallel (\alpha \frown \langle y\rangle)$	CSI7
$\mathsf{S_D}(x \trianglelefteq f?m \trianglerighteq y) = \mathsf{S_D}(x) \trianglelefteq f?m \trianglerighteq \mathsf{S_D}(y)$	S2D5
$\mathsf{S_D}(x \trianglelefteq f??m \trianglerighteq y) = \mathsf{S_D}(x) \trianglelefteq f??m \trianglerighteq \mathsf{S_D}(y)$	S2D6

Table 10. Additional transition rules for cyclic interleaving & deadlock at termination

$$\frac{\langle x_1,\rho\rangle\!\downarrow,\ldots,\langle x_k,\rho\rangle\!\downarrow,\langle x_{k+1},\rho\rangle \xrightarrow{\gamma} \langle x'_{k+1},\rho'\rangle}{\langle \parallel (\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha),\rho\rangle \xrightarrow{\gamma} \langle \parallel (\langle x'_{k+1}\rangle \frown \alpha),\rho'\rangle} (\alpha,\mathsf{T}) \in \rho(\langle\,\rangle) \quad (k \geq 0)$$

$$\frac{\langle x_1,\rho\rangle \not\rightarrow,\ldots,\langle x_k,\rho\rangle \not\rightarrow,\langle x_l,\rho\rangle\!\uparrow,\langle x_{k+1},\rho\rangle \xrightarrow{\gamma} \langle x'_{k+1},\rho'\rangle}{\langle \parallel (\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha),\rho\rangle \xrightarrow{\gamma} \langle \parallel (\langle x'_{k+1}\rangle \frown \alpha \frown \langle \mathsf{D}\rangle),\rho'\rangle} (\alpha,\mathsf{T}) \in \rho(\langle\,\rangle) \quad (k \geq l > 0)$$

$$\frac{\langle x_1,\rho\rangle\!\downarrow,\ldots,\langle x_k,\rho\rangle\!\downarrow,\langle x_{k+1},\rho\rangle \xrightarrow{\gamma} \langle x'_{k+1},\rho'\rangle}{\langle \parallel (\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha),\rho\rangle \xrightarrow{\gamma} \langle \parallel (\alpha \frown \langle x'_{k+1}\rangle),\rho'\rangle} (\alpha,\mathsf{F}) \in \rho(\langle\,\rangle) \quad (k \geq 0)$$

$$\frac{\langle x_1,\rho\rangle \not\rightarrow,\ldots,\langle x_k,\rho\rangle \not\rightarrow,\langle x_l,\rho\rangle\!\uparrow,\langle x_{k+1},\rho\rangle \xrightarrow{\gamma} \langle x'_{k+1},\rho'\rangle}{\langle \parallel (\langle x_1\rangle \frown \ldots \frown \langle x_{k+1}\rangle \frown \alpha),\rho\rangle \xrightarrow{\gamma} \langle \parallel (\alpha \frown \langle \mathsf{D}\rangle \frown \langle x'_{k+1}\rangle),\rho'\rangle} (\alpha,\mathsf{F}) \in \rho(\langle\,\rangle) \quad (k \geq l > 0)$$

$$\frac{\langle x,\rho\rangle \xrightarrow{\gamma} \langle x',\rho'\rangle}{\langle \mathsf{S_D}(x),\rho\rangle \xrightarrow{\gamma} \langle \mathsf{S_D}(x'),\rho'\rangle}$$

state that:

- after a positive reply on $f?m$ or $f??m$, the same thread proceeds with its next basic action; and thus it is prevented that meanwhile other threads can cause a state change to a state in which the processing of m is blocked (and $f?m$ would not reply positively) or the processing of m is refused (and both $f?m$ and $f??m$ would not reply positively);
- after a negative reply on $f?m$ or $f??m$, the same thread does not proceed with it; and thus it is prevented that other threads cannot make progress.

Without this difference, the Simulation Lemma (Section 7) would not go through.

The additional transition rules for cyclic interleaving and deadlock at termination in the presence of guarding tests are given in Table 10, where γ stands for an arbitrary basic action from the set $\{f?m, f??m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$.

A service may be local to a single thread, local to a multi-thread, local to a host, or local to a network. A service local to a multi-thread is shared by all threads from which the multi-thread is composed, etc. Henceforth, to simplify matters, it is assumed that each thread, each multi-thread, each host, and each network has a unique local service. Moreover, it is assumed that $\mathsf{t}, \mathsf{p}, \mathsf{h}, \mathsf{n} \in \mathcal{F}$. Below, the foci t, p, h and n play a special role:

- for each thread, t is the focus of its unique local service;
- for each multi-thread, p is the focus of its unique local service;
- for each host, h is the focus of its unique local service;

Table 11. Additional axioms for thread-service composition

$(x \trianglelefteq g?m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g?m \trianglerighteq (y /_f H)$	if $f \neq g$	TSC8
$(x \trianglelefteq f?m \trianglerighteq y) /_f H = \mathsf{tau} \circ (x /_f H)$	if $H(\langle m \rangle) \in \{\mathsf{T}, \mathsf{F}\}$	TSC9
$(x \trianglelefteq f?m \trianglerighteq y) /_f H = \mathsf{tau} \circ (y /_f H)$	if $H(\langle m \rangle) = \mathsf{B} \wedge f \neq \mathsf{t}$	TSC10
$(x \trianglelefteq f?m \trianglerighteq y) /_f H = \mathsf{D}$	if $(H(\langle m \rangle) = \mathsf{B} \wedge f = \mathsf{t}) \vee$ $H(\langle m \rangle) = \mathsf{R}$	TSC11
$(x \trianglelefteq g??m \trianglerighteq y) /_f H = (x /_f H) \trianglelefteq g??m \trianglerighteq (y /_f H)$	if $f \neq g$	TSC12
$(x \trianglelefteq f??m \trianglerighteq y) /_f H = \mathsf{tau} \circ (x /_f H)$	if $H(\langle m \rangle) \in \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$	TSC13
$(x \trianglelefteq f??m \trianglerighteq y) /_f H = \mathsf{tau} \circ (y /_f H)$	if $H(\langle m \rangle) = \mathsf{R}$	TSC14

Table 12. Additional transition rules for thread-service composition

$$\frac{\langle x, \rho\rangle \xrightarrow{f?m} \langle x', \rho'\rangle}{\langle x /_f H, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x' /_f H, \rho'\rangle} \; H(\langle m\rangle) \in \{\mathsf{T},\mathsf{F}\},\ (f?m, \mathsf{T}) \in \rho(\langle\,\rangle)$$

$$\frac{\langle x, \rho\rangle \xrightarrow{f?m} \langle x', \rho'\rangle}{\langle x /_f H, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x' /_f H, \rho'\rangle} \; H(\langle m\rangle) = \mathsf{B},\ f \neq \mathsf{t},\ (f?m, \mathsf{F}) \in \rho(\langle\,\rangle)$$

$$\frac{\langle x, \rho\rangle \xrightarrow{\mathsf{t}?m} \langle x', \rho'\rangle}{\langle x /_{\mathsf{t}} H, \rho\rangle\!\uparrow} \; H(\langle m\rangle) = \mathsf{B} \qquad \frac{\langle x, \rho\rangle \xrightarrow{f?m} \langle x', \rho'\rangle}{\langle x /_f H, \rho\rangle\!\uparrow} \; H(\langle m\rangle) = \mathsf{R}$$

$$\frac{\langle x, \rho\rangle \xrightarrow{f??m} \langle x', \rho'\rangle}{\langle x /_f H, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x' /_f H, \rho'\rangle} \; H(\langle m\rangle) \in \{\mathsf{T},\mathsf{F},\mathsf{B}\},\ (f?m, \mathsf{T}) \in \rho(\langle\,\rangle)$$

$$\frac{\langle x, \rho\rangle \xrightarrow{f??m} \langle x', \rho'\rangle}{\langle x /_f H, \rho\rangle \xrightarrow{\mathsf{tau}} \langle x' /_f H, \rho'\rangle} \; H(\langle m\rangle) = \mathsf{R},\ (f?m, \mathsf{F}) \in \rho(\langle\,\rangle)$$

– for each network, n is the focus of its unique local service.

The additional axioms for thread-service composition in the presence of guarding tests are given in Table 11. Axioms TSC10 and TSC11 are crucial. If $f = \mathsf{t}$, then f is the focus of the local service of the thread $x \trianglelefteq f?m \trianglerighteq y$. No other thread can raise a state of its local service in which the processing of m is blocked. Hence, if the processing of m is blocked, it is blocked forever.

The additional transition rules for thread-service composition in the presence of guarding tests are given in Table 12.

6 Delays and Exception Handling

We go on to show how guarding tests can used to express postconditional composition with delay and postconditional composition with exception handling.

For postconditional composition with delay, we extend the set of basic actions $\mathcal{A}$ with the set $\{f!m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. Performing a basic action $f!m$ is like performing $f.m$, but in case processing of the command m is temporarily blocked, it is automatically delayed until the blockade is over.

Table 13. Defining equation for postconditional composition with delay

$$x \trianglelefteq f!m \trianglerighteq y = (x \trianglelefteq f.m \trianglerighteq y) \trianglelefteq f?m \trianglerighteq (x \trianglelefteq f!m \trianglerighteq y)$$

Table 14. Defining equations for postconditional composition with exception handling

$$x \trianglelefteq f.m\,[y] \trianglerighteq z = (x \trianglelefteq f.m \trianglerighteq z) \trianglelefteq f??m \trianglerighteq y$$
$$x \trianglelefteq f!m\,[y] \trianglerighteq z = ((x \trianglelefteq f.m \trianglerighteq z) \trianglelefteq f?m \trianglerighteq (x \trianglelefteq f!m\,[y] \trianglerighteq z)) \trianglelefteq f??m \trianglerighteq y$$

Postconditional composition with delay is defined by the equation given in Table 13. The equation from this table guarantees that $f.m$ is only performed if $f?m$ yields a positive reply.

For postconditional composition with exception handling, we introduce the following notations: $x \trianglelefteq f.m\,[y] \trianglerighteq z$ and $x \trianglelefteq f!m\,[y] \trianglerighteq z$.

The intuition for $x \trianglelefteq f.m\,[y] \trianglerighteq z$ is that $x \trianglelefteq f.m \trianglerighteq z$ is tried, but y is done instead in the exceptional case that $x \trianglelefteq f.m \trianglerighteq z$ fails because the request to process m is refused. The intuition for $x \trianglelefteq f!m\,[y] \trianglerighteq z$ is that $x \trianglelefteq f!m \trianglerighteq z$ is tried, but y is done instead in the exceptional case that $x \trianglelefteq f!m \trianglerighteq z$ fails because the request to process m is refused. The processing of m may first be blocked and thereafter be refused; in that case, y is done instead as well.

The two forms of postconditional composition with exception handling are defined by the equations given in Table 14. The equations from this table guarantee that $f.m$ is only performed if $f?m$ yields a positive reply.

An alternative to the second equation from Table 14 is

$$x \trianglelefteq f!m\,[y] \trianglerighteq z = ((x \trianglelefteq f.m \trianglerighteq z) \trianglelefteq f?m \trianglerighteq (x \trianglelefteq f!m \trianglerighteq z)) \trianglelefteq f??m \trianglerighteq y\ .$$

In that case, y is only done if the processing of m is refused immediately.

7 A Formal Design Prototype

In this section, we show how the thread algebra developed so far can be used to give a simplified, formal representation schema of the design of systems that consist of several multi-threaded programs on various hosts in different networks. We propose to use the term *formal design prototype* for such a schema. The presented schema can be useful in understanding certain aspects of the system designed.

The set of *basic thread expressions*, with typical element P, is defined by

$$\begin{aligned} P \;::=\; & \mathsf{D} \mid \mathsf{S} \mid P \trianglelefteq f.m \trianglerighteq P \mid P \trianglelefteq f!m \trianglerighteq P \mid \\ & P \trianglelefteq f.m\,[P] \trianglerighteq P \mid P \trianglelefteq f!m\,[P] \trianglerighteq P \mid X_E\ , \end{aligned}$$

where $f \in \mathcal{F}$, $m \in \mathcal{M}$ and X_E is a constant standing for the unique solution for variable X of a guarded system of recursion equations E.

A thread vector in which each thread has its local service is of the form

$$\langle P_1 \between_{\mathsf{t}} TLS \rangle \frown \ldots \frown \langle P_n \between_{\mathsf{t}} TLS \rangle\ ,$$

where $P_1, \ldots, P_n$ are basic thread expressions and TLS is a local service for threads. TLS does nothing else but maintaining local data for a thread. A multi-thread vector in which each multi-thread has its local service is of the form

$$\langle \| (TV_1) /_{\mathsf{p}} PLS \rangle \frown \ldots \frown \langle \| (TV_m) /_{\mathsf{p}} PLS \rangle \ ,$$

where $TV_1, \ldots, TV_m$ are thread vectors in which each thread has its local service and PLS is a local service for multi-threads. PLS maintains shared data of the threads from which a multi-thread is composed. A typical example of such data are Java pipes. A host behavior vector in which each host has its local service is of the form

$$\langle \| (PV_1) /_{\mathsf{h}} HLS \rangle \frown \ldots \frown \langle \| (PV_l) /_{\mathsf{h}} HLS \rangle \ ,$$

where $PV_1, \ldots, PV_l$ are multi-thread vectors in which each multi-thread has its local service and HLS is a local service for hosts. HLS maintains shared data of the multi-threads on a host. A typical example of such data are the files connected with Unix sockets used for data transfer between multi-threads on the same host. A network behavior vector in which each network has its local service is of the form

$$\langle \| (HV_1) /_{\mathsf{n}} NLS \rangle \frown \ldots \frown \langle \| (HV_k) /_{\mathsf{n}} NLS \rangle \ ,$$

where $HV_1, \ldots, HV_k$ are host behavior vectors in which each host has its local service and NLS is a local service for networks. NLS maintains shared data of the hosts in a network. A typical example of such data are the files connected with Unix sockets used for data transfer between different hosts in the same network.

The behavior of a system that consist of several multi-threaded programs on various hosts in different networks is described by an expression of the form $\| (NV)$, where NV is a network behavior vector in which each network has its local service. A typical example is the case where NV is an expression of the form

$$\begin{aligned} \| (\langle \| (\langle \| (\langle P_1 /_{\mathsf{t}} TLS \rangle \frown \langle P_2 /_{\mathsf{t}} TLS \rangle) /_{\mathsf{p}} PLS \rangle \frown \\ \langle \| (\langle P_3 /_{\mathsf{t}} TLS \rangle \frown \langle P_4 /_{\mathsf{t}} TLS \rangle \frown \langle P_5 /_{\mathsf{t}} TLS \rangle) /_{\mathsf{p}} PLS \rangle) /_{\mathsf{h}} HLS \rangle \frown \\ \langle \| (\langle \| (\langle P_6 /_{\mathsf{t}} TLS \rangle) /_{\mathsf{p}} PLS \rangle) /_{\mathsf{h}} HLS \rangle) /_{\mathsf{n}} NLS \ , \end{aligned}$$

where $P_1, \ldots, P_6$ are basic thread expressions, and TLS, PLS, HLS and NLS are local services for threads, multi-threads, hosts and networks, respectively. It describes a system that consists of two hosts in one network, where on the first host currently a multi-thread with two threads and a multi-thread with three threads exist concurrently, and on the second host currently a single multi-thread with a single thread exists.

Evidence of correctness of the schema $\| (NV)$ is obtained by Lemma 1 given below. This lemma is phrased in terms of a simulation relation sim on the closed terms of the thread algebra developed in the preceding sections. The relation sim (is simulated by) is defined inductively by means of the rules in Table 15.

Table 15. Definition of simulation relation

$\mathsf{S}\ \mathsf{sim}\ x$
$\mathsf{D}\ \mathsf{sim}\ x$
$x\ \mathsf{sim}\ y \wedge x\ \mathsf{sim}\ z \Rightarrow x\ \mathsf{sim}\ y \trianglelefteq a \trianglerighteq z$
$x\ \mathsf{sim}\ y \wedge z\ \mathsf{sim}\ w \Rightarrow x \trianglelefteq a \trianglerighteq z\ \mathsf{sim}\ y \trianglelefteq a \trianglerighteq w$

Lemma 1 (Simulation Lemma). *Let P be a basic thread expression in which all basic actions are from the set $\{f.m \mid f \in \mathcal{F} \setminus \{\mathsf{t}, \mathsf{p}, \mathsf{h}, \mathsf{n}\}, m \in \mathcal{M}\}$ and constants standing for the solutions of guarded systems of recursion equations do not occur. Let $C[P]$ be a context of P of the form $\parallel (NV)$ where NV is a network behavior vector as above. Then $P\ \mathsf{sim}\ C[P]$. This implies that $C[P]$ will perform all steps of P in finite time.*

Proof. First we prove $P\ \mathsf{sim}\ C'[P]$, where C' is a context of P of the form $\parallel (TV)$, by induction on the depth of P, and in both the basis and the inductive step, by induction on the position of P in thread vector TV. Using in each case the preceding result, we prove an analogous result for each higher-level vector in a similar way.

8 Conclusions

We have presented an algebraic theory of threads and multi-threads based on multi-level strategic interleaving for the simple strategy of cyclic interleaving. The other interleaving strategies treated in [4] can be adapted to the setting of multi-level strategic interleaving in a similar way. We have also presented a reasonable though simplified formal representation schema of the design of systems that consist of several multi-threaded programs on various hosts in different networks. By dealing with delays and exceptions, this schema is sufficiently expressive to formalize mechanisms like Java pipes (for communication between threads) and Unix sockets (for communication between multi-threads, called processes in Unix jargon, and communication between hosts). The exception handling notation introduced is only used for single threads and a translation takes care of its meaning.

References

1. Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley (1996)
2. Bishop, J., Horspool, N.: C# Concisely. Addison-Wesley (2004)
3. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control **60** (1984) 109–137
4. Bergstra, J.A., Middelburg, C.A.: Thread algebra for strategic interleaving. Computer Science Report 04-35, Department of Mathematics and Computer Science, Eindhoven University of Technology (2004)

5. Bergstra, J.A., Loots, M.E.: Program algebra for sequential code. Journal of Logic and Algebraic Programming **51** (2002) 125–156
6. Bergstra, J.A., Bethke, I.: Polarized process algebra and program equivalence. In Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J., eds.: Proceedings 30th ICALP. Volume 2719 of Lecture Notes in Computer Science., Springer-Verlag (2003) 1–21
7. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural operational semantics. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier, Amsterdam (2001) 197–292
8. Middelburg, C.A.: An alternative formulation of operational conservativity with binding terms. Journal of Logic and Algebraic Programming **55** (2003) 1–19