Distributed Strategic Interleaving with Load Balancing

J. A. Bergstra [a,b], C. A. Middelburg [a,*,1]

[a] *Programming Research Group, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, the Netherlands*

[b] *Department of Philosophy, Utrecht University, P.O. Box 80126, 3508 TC Utrecht, the Netherlands*

Abstract

In a previous paper, we developed an algebraic theory of threads, interleaving of threads, and interaction of threads with services. In the current paper, we assume that the threads and services are distributed over the nodes of a network. We extend the theory developed so far to the distributed case by introducing distributed interleaving strategies that support explicit thread migration and see to load balancing or capability searching by implicit thread migration. The extension to the distributed case provides insight into details of multi-threading that come up in a networked environment.

Key words: thread algebra, distributed strategic interleaving, thread migration, load balancing, capability searching

1 Introduction

The objective of this paper is to develop an algebraic theory of threads, interleaving of threads, and interaction of threads with services that covers the case where the threads and services are distributed over the nodes of a network and both explicit thread migration and implicit thread migration take place.

* Corresponding author. Tel: +31 20 525 7583; fax: +31 20 525 7490.
Email addresses: J.A.Bergstra@uva.nl (J. A. Bergstra), C.A.Middelburg@uva.nl (C. A. Middelburg).

[1] This research was partly carried out while the second author was also at Eindhoven University of Technology, Division of Computer Science.

A thread is the behaviour 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 contemporary object-oriented programming languages such as Java [17] and C# [18]. Arbitrary interleaving, on which theories about concurrent processes such as ACP [2,14], CCS [20] and CSP [19] are based, is not an appropriate abstraction when dealing with multi-threading. In the case of multi-threading, some deterministic interleaving strategy is used. In [6], we introduced a number of plausible deterministic interleaving strategies for multi-threading. We proposed to use the phrase strategic interleaving for the more constrained form of interleaving obtained by using such a strategy. Services are the components of the execution environment of a thread with which the thread interacts. Services can process actions performed by a thread and return replies to the thread. In [6], we also introduced a feature for interaction of threads with services. The algebraic theory of threads, interleaving of threads, and interaction of threads with services is called thread algebra.

In the current paper, we assume that the threads and services are distributed over the nodes of a network. We extend the theory developed so far to the distributed case by introducing distributed interleaving strategies that support explicit thread migration and see to load balancing or capability searching by implicit thread migration. These distributed interleaving strategies are adaptations of the simplest interleaving strategy introduced in [6], to wit pure cyclic interleaving,[2] to the distributed case. The other interleaving strategies from [6] can be adapted in a similar way. Load balancing and capability searching are considered implicit forms of thread migration that may be part of distributed interleaving strategies. Load balancing is intended for equalizing the execution speed of the different threads that exist concurrently in a distributed system. Capability searching is intended for achieving that the different threads that exist concurrently in a distributed system are executed where the services needed are available.

A main assumption made in this paper is that, in the case where systems are networks of nodes over which threads and services are distributed, a single interleaving strategy determines how the threads that exist concurrently at the different nodes are interleaved. This is a drastic simplification, as a result of which intuition may break down. We believe however that some such simplification is needed to obtain a manageable theory about the behaviour of such systems – and that the resulting theory will sometimes be adequate and sometimes be inadequate. Moreover, cyclic interleaving is a simplification of the interleaving strategies actually used. Because of the complexity of those strategies, we consider a simplification like this one desirable to start with.

[2] Implementations of the pure cyclic interleaving strategy are usually called round-robin schedulers.

Our work on thread algebra marks itself off from much work on the subject of multi-threading because we abandon the usual point of view that arbitrary interleaving is an appropriate abstraction when dealing with multi-threading. The following points illustrate why that point of view is untenable: (i) whether the interleaving of certain threads leads to deadlock depends on the interleaving strategy used; (ii) sometimes deadlock takes place with a particular interleaving strategy whereas arbitrary interleaving would not lead to deadlock and vice versa. We give demonstrations of (i) and (ii) in [6] and [8], respectively.

The thread-service dichotomy made in thread algebra is useful for the following reasons: (i) for services, a state-based description is generally more convenient than an action-based description whereas it is the other way round for threads; (ii) the interaction of threads with services is of an asymmetric nature. In [8], evidence of both (i) and (ii) is produced by the established connections of threads and services with processes as considered in an extension of ACP with conditions introduced in [7].

Thread algebra is a design on top of an algebraic theory of the behaviour of deterministic sequential programs under execution introduced in [5] under the name Basic Polarized Process Algebra (BPPA). Prompted by the development of thread algebra, it has been renamed to Basic Thread Algebra (BTA).

This paper is organized as follows. Before we take up interleaving strategies for threads that are distributed over the nodes of a network, we review BTA and guarded recursion in the setting of BTA (Sections 2 and 3). We also discuss the approximation induction principle in this setting (Section 4). After that, we introduce a very simple interleaving strategy for threads that are distributed over the nodes of a network which supports explicit thread migration (Section 5). Then, we provide a classification of services that will be used in subsequent sections (Section 6). Following this, we introduce a variation of the very simple interleaving strategy that prevents threads from migrating while they keep locks on shared services (Section 7). Thereupon, we introduce three variations of the second interleaving strategy that see to load balancing by means of implicit migration (Sections 8, 9 and 10). Next, we introduce a variation of the second interleaving strategy that takes care of capability searching by means of implicit migration (Section 11). After that, we introduce thread-service composition to allow for threads to be affected by services (Section 12). Then, because it adds to the need for load balancing, we introduce a basic form of thread forking (Section 13). Finally, we make some concluding remarks and mention some options for future work (Section 14).

2 Basic Thread Algebra

In this section, we review BTA (Basic Thread Algebra), introduced in [5] under the name BPPA (Basic Polarized Process Algebra). BTA is a form of process algebra which is tailored to the description of the behaviour of deterministic sequential programs under execution. The behaviours concerned are called *threads*.

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

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

Although BTA is one-sorted, we make this sort explicit. The reason for this is that we will extend BTA with additional sorts in Section 5.

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

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

Terms of sort $\mathbf{T}$ are built as usual. Throughout the paper, we assume that there are infinitely many variables of sort $\mathbf{T}$, including x, y, z.

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

Let p and q be closed terms of sort $\mathbf{T}$ and $a \in \mathcal{A}_\tau$. Then $p \trianglelefteq a \trianglerighteq q$ will perform action a, and after that proceed as p if the processing of a leads to the reply T (called a positive reply) and proceed as q if the processing of a leads to the reply F (called a negative reply). The action tau plays a special role. Its execution will never change any state and always lead to a positive reply.

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

Table 1
Axiom of BTA

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

Table 2
Axioms for guarded recursion

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

The structural operational semantics of BTA can be found in [6,10].

3 Guarded Recursion

Each closed term of sort **T** from the language of BTA denotes a finite thread, i.e. a thread of which the length of the sequences of actions that it can perform is bounded. Guarded recursion gives rise to infinite threads. In this section, we extend BTA with guarded recursion.

A *guarded recursive specification* over BTA is a set of recursion equations $E = \{X = t_X \mid X \in V\}$, where V is a set of variables of sort **T** and each t_X is a term of the form D, S or $t \trianglelefteq a \trianglerighteq t'$ with t and t' terms of sort **T** from the language of BTA that contain only variables from V. We write $\mathrm{V}(E)$ for the set of all variables that occur on the left-hand side of an equation in E. We are only interested in models of BTA in which guarded recursive specifications have unique solutions, such as the projective limit model of BTA presented in [3].

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

Table 3
Approximation induction principle

$\bigwedge_{n\geq 0} \pi_n(x) = \pi_n(y) \Rightarrow x = y$	AIP

Table 4
Axioms for projection operators

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

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

4 Approximation Induction Principle

Closed terms of sort **T** from the language of BTA+REC that denote the same infinite thread cannot always be proved equal by means of the axioms of BTA+REC. In this section, we introduce the approximation induction principle to remedy this.

The approximation induction principle, AIP in short, is based on the view that two threads are identical if their approximations up to any finite depth are identical. The approximation up to depth n of a thread is obtained by cutting it off after performing a sequence of actions of length n.

AIP is the infinitary conditional equation given in Table 3. Here, following [5], approximation of depth n is phrased in terms of a unary *projection* operator π_n. The axioms for the projection operators are given in Table 4. In this table, a stands for an arbitrary member of $\mathcal{A}_\tau$.

Let T stand for either BTA, BTA+REC or any other extension of BTA introduced in this paper. Then we will write T+AIP for T extended with the projections operators π_n and the axioms P0–P3 and AIP.

RDP, RSP and AIP originate from work on ACP [2]. In the setting of ACP, these principles were first formulated in [4]. Like in the setting of ACP, RSP follows from RDP and AIP.

5 Distributed Strategic Interleaving of Threads

In this section, we take up the extension of BTA to a theory about threads and distributed multi-threading by introducing a very simple distributed interleaving strategy. The resulting theory is called $\mathrm{TA}_{\mathrm{dsi}}$.

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

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

Threads that perform unlocated actions only are called *unlocated* threads and threads that perform located actions only are called *located* threads. The behaviour of a distributed system of the kind considered in this paper is supposed to be a located thread. However, the threads that exist concurrently at the different locations in the system are unlocated threads.

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

Distributed strategic interleaving operators turn a distributed thread vector of arbitrary length into a single located thread. This single located thread obtained via a distributed strategic interleaving operator is also called a distributed multi-thread. In this paper, we cover adaptations of the simplest interleaving strategy, namely pure *cyclic interleaving*, which support explicit thread migration and see to load balancing or capability searching by implicit thread migration.

In the distributed case, cyclic interleaving basically operates as follows: at each stage of the interleaving, the first thread in the first local thread vector in the distributed thread vector gets a turn to perform an action at the location at which the first local thread vector is and then the distributed thread vector undergoes cyclic permutation. We mean by cyclic permutation of a distributed

thread vector that the first thread in the first local thread vector becomes the last thread in the first local thread vector, all other threads in the first local thread vector move one position to the left, the resulting local thread vector becomes the last local thread vector in the distributed thread vector, and all other local thread vectors in the distributed thread vector move one position to the left. If a thread in one of the local thread vectors deadlocks, the whole does not deadlock till all others have terminated or deadlocked. An important property of this interleaving strategy is that it is fair in the sense that there will always come a next turn for all active threads. This is a strong kind of fairness. A weaker kind of fairness would for instance arise if there will always come a next turn for all locations with active threads, but not for all active threads at each of those locations.

Other plausible interleaving strategies for the case where threads and services are not distributed are treated in [6]. They can also be adapted to the case where threads and services are distributed.

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

$\mathrm{TA}_{\mathrm{dsi}}$ has the sort $\mathbf{T}$ of BTA and in addition the following sorts:

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

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

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

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

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

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

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

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

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

Terms of the different sorts are built as usual for a many-sorted signature (see e.g. [21,23]). Throughout the paper, we assume that there are infinitely many variables of sort **LT**, including u, v, w, infinitely many variables of sort **LTV**, including $\alpha, \alpha_1, \alpha_2$, and infinitely many variables of sort **DTV**, including β, β_1, β_2.

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

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

Essentially, the sort **DTV** includes all sequences of pairs consisting of a location and a local thread vector. The ones that contain a unique pair for each location are the proper distributed thread vectors in the sense that the cyclic distributed interleaving strategy outlined above is intended for them. Improper distributed thread vectors that do not contain duplicate pairs for some location are needed in the axiomatization of this strategy. Improper distributed thread vectors that do contain duplicate pairs for some location appear to have more than one local thread vector at the location concerned. Their exclusion would make it necessary for concatenation of distributed thread vectors to be turned into a partial operator. The cyclic distributed interleaving strategy never turns a proper distributed thread vector into an improper one or the other way round. Similar remarks apply to the enriched distributed thread vectors introduced in subsequent sections and will not be repeated.

Let p and q be closed terms of sort **LT** from the language of $\mathrm{TA}_{\mathrm{dsi}}$, $l \in \mathcal{L}$ and $a \in \mathcal{A}_\tau$. Then $p \trianglelefteq l.a \trianglerighteq q$ will perform action a at location l, and after that proceed as p if the processing of a leads to the reply T (called a positive reply) and proceed as q if the processing of a leads to the reply F (called a negative reply).

Table 5
Definition of the functions app_l

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

Table 6
Axioms for postconditional composition

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

Table 7
Axioms for deadlock at termination

$\mathsf{S_D}(\mathsf{S}) = \mathsf{D}$	S2D1
$\mathsf{S_D}(\mathsf{D}) = \mathsf{D}$	S2D2
$\mathsf{S_D}(l.\mathsf{tau} \circ u) = l.\mathsf{tau} \circ \mathsf{S_D}(u)$	S2D3
$\mathsf{S_D}(u \trianglelefteq l.a \trianglerighteq v) = \mathsf{S_D}(u) \trianglelefteq l.a \trianglerighteq \mathsf{S_D}(v)$	S2D4

The deadlock at termination operator $\mathsf{S_D}$ is an auxiliary operator used in the axioms for cyclic distributed interleaving introduced below to turn termination into deadlock. Let p be a closed term of sort **LT** from the language of $\mathrm{TA}_{\mathrm{dsi}}$. Then $\mathsf{S_D}(p)$ behaves the same as p, except that it deadlocks whenever p would terminate.

The cyclic distributed strategic interleaving operator serves for interleaving of the threads in a proper distributed thread vector according to the strategy outlined above, but with support of explicit thread migration. In the case where a local thread vector of the form $\langle p \trianglelefteq \mathsf{mg}(n) \trianglerighteq q\rangle \frown \alpha$ with $n \in \mathcal{L}$ is encountered as the first local thread vector, α becomes the last local thread vector in the distributed thread vector and p is appended to the local thread vector at location n. If $n \notin \mathcal{L}$, then $\alpha \frown \langle q\rangle$ becomes the last local thread vector in the distributed thread vector. The migration postconditional composition operators have the same shape as the postconditional composition operators introduced earlier. However, formally no action is involved in migration postconditional composition.

In the axioms for cyclic distributed interleaving introduced below, a binary function app_l ($l \in \mathcal{L}$) from unlocated threads and distributed thread vectors to distributed thread vectors is used which maps each unlocated thread x and distributed thread vector β to the distributed thread vector obtained by appending x to the local thread vector at location l in β. The functions app_l are defined in Table 5.

The axioms for postconditional composition on located threads, deadlock at termination and cyclic distributed interleaving are given in Tables 6, 7 and 8. In these tables, a stands for an arbitrary action from $\mathcal{A}$. In these tables and all subsequent tables with axioms for a distributed interleaving strategy, $l_1, \ldots, l_k$ and l stand for arbitrary locations from $\mathcal{L}$ and n stands for an arbitrary natural

Table 8
Axioms for cyclic distributed interleaving

$\|(\langle\rangle) = \mathsf{S}$	CDI1
$\|([\langle\rangle]_{l_1} \frown \ldots \frown [\langle\rangle]_{l_k}) = \mathsf{S}$	CDI2
$\|([\langle\rangle]_l \frown \beta) = \|(\beta \frown [\langle\rangle]_l)$	CDI3
$\|([\langle \mathsf{S}\rangle \frown \alpha]_l \frown \beta) = \|(\beta \frown [\alpha]_l)$	CDI4
$\|([\langle \mathsf{D}\rangle \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l))$	CDI5
$\|([\langle \mathsf{tau} \circ x\rangle \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(\beta \frown [\alpha \frown \langle x\rangle]_l)$	CDI6
$\|([\langle x \trianglelefteq a \trianglerighteq y\rangle \frown \alpha]_l \frown \beta) = \|(\beta \frown [\alpha \frown \langle x\rangle]_l) \trianglelefteq l.a \trianglerighteq \|(\beta \frown [\alpha \frown \langle y\rangle]_l)$	CDI7
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(app_n(x, \beta \frown [\alpha]_l))$ if $n \in \mathcal{L}$	CDI8
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(\beta \frown [\alpha \frown \langle y\rangle]_l)$ if $n \notin \mathcal{L}$	CDI9

number. Axioms CDI1–CDI9 set out the details of the simple cyclic distributed interleaving strategy introduced in this section in a clear and concise way.

To be fully precise, we should give axioms concerning the constants and operators to build terms of the sorts **LTV** and **DTV** as well. We refrain from doing so because the constants and operators concerned are the usual ones for sequences – the singleton distributed thread vector operators involve an implicit pairing of their operand with a location. [3] Similar remarks apply to the sorts of local thread vectors and distributed thread vectors introduced in subsequent sections and will not be repeated.

Guarded recursion can be added to $\mathrm{TA}_{\mathrm{dsi}}$ as it is added to BTA in Section 3. We will write $\mathrm{TA}_{\mathrm{dsi}}$+REC for the resulting theory. The merit of $\mathrm{TA}_{\mathrm{dsi}}$+REC is that it covers cyclic distributed interleaving of threads in which infinite threads are involved.

The axioms for cyclic distributed strategic interleaving given in Table 8 constitute a definition by recursion on the sum of the depths of all threads in the distributed thread vector with case distinction on the structure of the first thread in the first local thread vector. Hence, it is obvious that the axioms are consistent and that every closed term of the sort **LT** from the language of $\mathrm{TA}_{\mathrm{dsi}}$ is derivably equal to one that does not contain the cyclic distributed strategic interleaving operator. Moreover, it follows easily that every projection of a closed term of the sort **LT** from the language of $\mathrm{TA}_{\mathrm{dsi}}$+REC is derivably equal to one that does not contain the cyclic distributed strategic interleaving operator. Similar remarks apply to the other distributed strategic interleaving operators introduced in this paper and will not be repeated.

We refrain from giving the structural operational semantics of $\mathrm{TA}_{\mathrm{dsi}}$. The transition rules for the postconditional composition operators on located threads constitute a trivial adaptation of the transition rules for the postconditional composition operators on unlocated threads. The transition rules for the dead-

[3] This amounts to looking upon $[\alpha]_l$ as syntactic sugar for $\langle(l, \alpha)\rangle$.

lock at termination operator can be found in [6,10]. The transition rules for the cyclic distributed strategic interleaving operator, as well as the transition rules for the distributed strategic interleaving operators that will be introduced later in this paper, are similar to the transition rules for the strategic interleaving operator for cyclic interleaving found in [6,10]. However, the transition rules for a distributed strategic interleaving operator are so much more involved that they might be called unreadable. This implies that they cannot add to a better understanding of the distributed interleaving strategy concerned.

Henceforth, we will write $\mathrm{TA}_{\mathrm{dsi}}(A)$ for $\mathrm{TA}_{\mathrm{dsi}}$ with the set of basic actions $\mathcal{A}$ fixed to be the set A.

6 A Classification of Services

In this section, we provide a useful classification of services present at each of the locations over which the threads and services of a system are distributed. This classification is a slightly adapted version of the classification given in [6] and will be used in subsequent sections.

A major distinction is between target services and para-target services:

- A service is a *target service* if the result of the processing of commands by the service is partly observable externally. Printing a document, sending an email message, showing data on a display or writing persistent data in permanent memory are typical examples of using a target service.
- A service is a *para-target service* if the result of the processing of commands by the service is wholly unobservable externally. Setting a timer or transferring data by means of a Java pipe are typical examples of using a para-target service.

In the case of para-target services, a further distinction is between private services and shared services:

- A *private service* provides its service to a single thread only. A timer is usually a private service. Private services exist to support a single thread in creating intended behaviour in relation to target services.
- A *shared service* provides its service to all threads existing at some location. A Java pipe is an example of a shared para-target service. Shared services exist to support all threads existing at the same location in creating intended behaviour in relation to target services.

To simplify matters, it will be assumed that all para-target services at some location are either private or shared, thus disregarding the possibility that

a service provides its service to some of the threads existing at the location only. This leaves us with execution architectures that provide target services, shared services and private services.

Private services are called local services in [6]. We use the phrase private service here because local service may be confusing in the current setting where services are located.

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

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

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

Henceforth, to simplify matters, it is assumed that each thread has a unique private service. This service has to move along with the thread if it migrates to another location.

7 Distributed Strategic Interleaving and Locking

In this section, we introduce a variation of the distributed interleaving strategy from Section 5 that prevents threads from migrating while they keep locks on shared services. This results in a theory called $\mathrm{TA}^{\mathrm{lck}}_{\mathrm{dsi}}$.

In order to deal with locking of shared services, 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* $\mathcal{M}$. 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. We write FM for the set $\{f.m \mid f \in \mathcal{F}, m \in \mathcal{M}\}$. For the set $\mathcal{A}$ of unlocated actions, we take the set FM.

Performing an unlocated action $f.m$ is taken as making a request to the service named f at a location still to be fixed by the distributed interleaving strategy to process command m. Performing a located action $l.f.m$ is taken as making a request to the service named f at location l to process command m.

It is also assumed that $\mathsf{t}, \mathsf{s}_1, \mathsf{s}_2, \ldots \in \mathcal{F}$. The foci t, s_1, s_2, ... play a special role:

- for each thread, t is the focus of the unique private service of that thread;
- for each location, s_1, s_2, ... are the foci of the services shared by all threads from the local thread vector at that location.

We write $\mathcal{F}_\mathsf{s}$ for the set $\{\mathsf{s}_1, \mathsf{s}_2, \ldots\}$ of shared service foci.

For each location, all threads from the local thread vector at that location are supposed to be able to lock and unlock the services with foci from $\mathcal{F}_\mathsf{s}$ at that location. It is assumed that $\mathsf{lock}, \mathsf{unlock} \in \mathcal{M}$. We write $\mathcal{M}_\mathsf{l}$ for the set $\{\mathsf{lock}, \mathsf{unlock}\}$ of locking methods.

When processing a command, the services with foci from $\mathcal{F}_\mathsf{s}$ do nothing but changing a state and producing replies on the basis of the state. In every state, either the processing of lock leads to a positive reply or the processing of unlock leads to a positive reply, but not both, and if the processing of one of them leads to a positive reply, then the processing concerned moves the service to a state in which the processing of the other one leads to a positive reply. In this way, successive lockings without intermediate unlocking is prevented. It is assumed that initially the processing of lock leads to a positive reply. Moreover, it is assumed that a request to process lock or unlock is rejected by services with foci other than foci from $\mathcal{F}_\mathsf{s}$. Rejecting a request to process a command brings with it deadlocking of the thread making the request.

If a thread successfully performs the action $\mathsf{s}_i.\mathsf{lock}$, it acquires the lock of the service with focus s_i. It keeps the lock until it is released by performing $\mathsf{s}_i.\mathsf{unlock}$. All threads are supposed to work as follows. For each focus s_i, all commands $\mathsf{s}_i.m$ different from $\mathsf{s}_i.\mathsf{lock}$ must be performed in a phase in which the thread keeps the lock of the service with focus s_i. If all threads adhere to this rule, it is guaranteed that a thread keeping the lock on a service has exclusive access to that service. The distributed interleaving strategy introduced in this section permits of no deviation from this rule.

Table 9
Definition of the functions app_l

$app_l(x, \langle\rangle) = \langle\rangle$
$app_l(x, [\alpha]_{l'} \frown \beta) = [\alpha \frown \langle x \rangle_\emptyset]_l \frown \beta$ if $l = l'$
$app_l(x, [\alpha]_{l'} \frown \beta) = [\alpha]_{l'} \frown app_l(x, \beta)$ if $l \neq l'$

In the presence of locking and unlocking of shared services, axiom CDI8 (Table 8) is not satisfactory. A thread from the local thread vector at one location should not be permitted to migrate to another location if the thread keeps the lock of one or more shared services at the former location.

To deal with that, we have to enrich distributed thread vectors by replacing each thread in each local thread vector by a pair consisting of the thread and the set of all foci naming shared services on which the thread keeps a lock.

We mention that, when the interleaving of the threads in a distributed thread vector makes a start, the threads are supposed to keep no lock.

$\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lck}}$ has the same sorts as $\mathrm{TA}_{\mathrm{dsi}}(FM)$. To build terms of the sorts **T**, **LT** and **DTV**, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lck}}$ has the same constants and operators as $\mathrm{TA}_{\mathrm{dsi}}(FM)$. To build terms of sort **LTV**, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lck}}$ has the following constants and operators:

- the *empty local thread vector* constant $\langle\,\rangle : \mathbf{LTV}$;
- for each $F \subseteq \mathcal{F}_\mathsf{s}$, the unary *singleton local thread vector* operator $\langle\_\rangle_F : \mathbf{T} \to \mathbf{LTV}$;
- the binary *local thread vector concatenation* operator $\frown : \mathbf{LTV} \times \mathbf{LTV} \to \mathbf{LTV}$.

That is, the operator $\langle\_\rangle$ is replaced by the operators $\langle\_\rangle_F$.

Similar to the singleton distributed thread vector operators $[\_]_l$, the singleton local thread vector operators $\langle\_\rangle_F$ involve an implicit pairing of their operand with a set of foci naming shared services.

In the axioms for cyclic distributed interleaving with locking introduced below, a binary function app_l ($l \in \mathcal{L}$) from unlocated threads and distributed thread vectors to distributed thread vectors is used which maps each unlocated thread x and distributed thread vector β to the distributed thread vector obtained by appending x to the local thread vector at location l in β and associating the empty set as set of foci with x. The functions app_l are defined in Table 9.

The axioms for cyclic distributed interleaving with locking are given in Table 10. In this table and all subsequent tables with axioms for a distributed interleaving strategy, f stands for an arbitrary focus from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$, and F stands for an arbitrary subset of $\mathcal{F}_\mathsf{s}$. Axiom CDI7 of the strategy from the previous section is replaced by four axioms to make distinction between lock actions, unlock actions and other actions. This

Table 10
Axioms for cyclic distributed interleaving with locking

$\parallel(\langle\rangle) = \mathsf{S}$	CDIlck1
$\parallel([\langle\rangle]_{l_1} \frown \ldots \frown [\langle\rangle]_{l_k}) = \mathsf{S}$	CDIlck2
$\parallel([\langle\rangle]_l \frown \beta) = \parallel(\beta \frown [\langle\rangle]_l)$	CDIlck3
$\parallel([\langle \mathsf{S}\rangle_F \frown \alpha]_l \frown \beta) = \parallel(\beta \frown [\alpha]_l)$	CDIlck4
$\parallel([\langle \mathsf{D}\rangle_F \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\parallel(\beta \frown [\alpha]_l))$	CDIlck5
$\parallel([\langle \mathsf{tau} \circ x\rangle_F \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \parallel(\beta \frown [\alpha \frown \langle x\rangle_F]_l)$	CDIlck6
$\parallel([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) =$	
$\quad \parallel(\beta \frown [\alpha \frown \langle x\rangle_F]_l) \trianglelefteq l.f.m \trianglerighteq \parallel(\beta \frown [\alpha \frown \langle y\rangle_F]_l)$	
$\quad \mathsf{if}\ f \notin \mathcal{F}_\mathsf{s} \vee (f \in F \wedge m \notin \mathcal{M}_\mathsf{l})$	CDIlck7a
$\parallel([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) =$	
$\quad \parallel(\beta \frown [\alpha \frown \langle x\rangle_{F\cup\{f\}}]_l) \trianglelefteq l.f.\mathsf{lock} \trianglerighteq \parallel(\beta \frown [\alpha \frown \langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F]_l)$	
$\quad \mathsf{if}\ f \in \mathcal{F}_\mathsf{s} \setminus F$	CDIlck7b
$\parallel([\langle x \trianglelefteq f.\mathsf{unlock} \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) =$	
$\quad \parallel(\beta \frown [\alpha \frown \langle x\rangle_{F\setminus\{f\}}]_l) \trianglelefteq l.f.\mathsf{unlock} \trianglerighteq \parallel(\beta \frown [\alpha \frown \langle y\rangle_F]_l)$	
$\quad \mathsf{if}\ f \in F$	CDIlck7c
$\parallel([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\parallel(\beta \frown [\alpha]_l))$	
$\quad \mathsf{if}\ f \in \mathcal{F}_\mathsf{s} \wedge (f \in F \vee m \neq \mathsf{lock}) \wedge (f \in \mathcal{F}_\mathsf{s} \setminus F \vee m = \mathsf{lock})$	CDIlck7d
$\parallel([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \parallel(app_n(x, \beta \frown [\alpha]_l))$ if $n \in \mathcal{L} \wedge F = \emptyset$	CDIlck8
$\parallel([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \parallel(\beta \frown [\alpha \frown \langle y\rangle_F]_l)$ if $n \notin \mathcal{L} \vee F \neq \emptyset$	CDIlck9

distinction must be made to keep track of the locks that the different threads keep and to preclude the possibility of deviation from the rule that threads must adhere to in order to guarantee exclusive access to locked services.

Guarded recursion can be added to $\mathrm{TA}^{\mathrm{lck}}_{\mathrm{dsi}}$ as it is added to BTA in Section 3.

8 Distributed Strategic Interleaving and Load Balancing

In this section, we introduce a variation of the distributed interleaving strategy from Section 7 with implicit migration of threads to achieve load-balancing. This results in a theory called $\mathrm{TA}^{\mathrm{lba}}_{\mathrm{dsi}}$.

Immediately after the current thread has performed an action, implicit migration of that thread to another location may take place. Whether migration really takes place, depends on three parameters of the strategy: k_1, k_2 and k_3. All three parameters are natural numbers greater than 0. The current thread is implicitly migrated if the following conditions are fulfilled:

- the current thread is migration compatible;
- the current thread keeps no locks;
- the number of steps that the current thread has performed at its current location is at least k_1;

- the largest number of steps that the current thread will perform before termination, deadlock or explicit migration is at least k_2;
- the length of the local thread vector that contains the current thread, say n, is greater than k_3 and there is another local thread vector whose length times 2 is less than n.

The purpose of these conditions is as follows:

- the first condition is to exclude migration of threads whose interaction with services is in whole or in part bound to services at the locations they are;
- the second condition is to prevent threads from migrating while they keep locks on shared services;
- the third condition is to guard threads against hopping around uselessly;
- the fourth condition is to preclude the possibility that threads are migrated while they will necessarily terminate, deadlock or migrate explicitly soon;
- the last condition is to achieve that threads from locations with long local thread vectors are migrated to locations with short local thread vectors.

If the current thread is implicitly migrated, then it will be migrated to the first among the locations with a shortest local thread vector.

To understand why load balancing strategic interleaving may be useful, assume that: (i) time is divided into slices of equal length; (ii) during each slice, all local thread vectors get a turn to perform an action of one thread. Thus, a long local thread vector gives rise to slow execution per thread whereas a short local thread vector gives rise to fast execution per thread. Load balancing will equalize the execution speed of the threads in the different local thread vectors.

To deal with both locking and load balancing, we have to enrich distributed thread vectors by replacing each thread in each local thread vector by a triple consisting of the thread, the set of all foci naming services on which the thread keeps a lock, and the minimum of k_1 and the number of steps that the thread has performed at the location it is.

We mention that, when the interleaving of the threads in a distributed thread vector makes a start, the threads are supposed to keep no lock and to have performed no steps at the location they are.

It is assumed that each method can be classified as either *functional* or *non-functional*, so that the set $\mathcal{M}_{\mathsf{fn}} \subseteq \mathcal{M}$ of functional methods can be distinguished. Moreover, it is assumed that $\mathsf{lock} \notin \mathcal{M}_{\mathsf{fn}}$ and $\mathsf{unlock} \notin \mathcal{M}_{\mathsf{fn}}$. A method is a functional method if its processing by a shared service never change the state of the shared service.

Migration compatibility is a criterion to exclude migration of threads whose interaction with services is in whole or in part bound to services at the location

Table 11
Definition of the functions app_l

$app_l(x, \langle\rangle) = \langle\rangle$
$app_l(x, [\alpha]_{l'} \frown \beta) = [\alpha \frown \langle x \rangle^0_\emptyset]_l \frown \beta$ if $l = l'$
$app_l(x, [\alpha]_{l'} \frown \beta) = [\alpha]_{l'} \frown app_l(x, \beta)$ if $l \neq l'$

they are. Interaction of threads with target services is always bound, interaction of threads with their private service is never bound, and interaction of threads with shared services using functional methods only is never bound. However, interaction of threads with shared services may be bound if non-functional methods are used. In this paper, a thread is considered migration compatible if its interaction with services includes nothing but interaction with private services and interaction with shared services using functional methods. This makes migration compatibility rather restrictive.

We can take a less restrictive criterion instead if we assume that locked sessions with shared services can also be classified as either functional or non-functional. The criterion in question is complicated in such a way that it would distract the attention from the main issues treated in this paper.

$\mathrm{TA}^{\mathrm{lba}}_{\mathrm{dsi}}$ has the same sorts as $\mathrm{TA}^{\mathrm{lck}}_{\mathrm{dsi}}$. To build terms of the sorts **T**, **LT** and **DTV**, $\mathrm{TA}^{\mathrm{lba}}_{\mathrm{dsi}}$ has the same constants and operators as $\mathrm{TA}^{\mathrm{lck}}_{\mathrm{dsi}}$. To build terms of sort **LTV**, $\mathrm{TA}^{\mathrm{lba}}_{\mathrm{dsi}}$ has the following constants and operators:

- the *empty local thread vector* constant $\langle\,\rangle : \mathbf{LTV}$;
- for each $F \subseteq \mathcal{F}_{\mathsf{s}}$ and $c \in \mathbb{N}$ such that $c \leq k_1$, the unary *singleton local thread vector* operator $\langle\_\rangle^c_F : \mathbf{T} \to \mathbf{LTV}$;
- the binary *local thread vector concatenation* operator $\frown : \mathbf{LTV} \times \mathbf{LTV} \to \mathbf{LTV}$.

That is, the operator $\langle\_\rangle_F$ is replaced by the operators $\langle\_\rangle^c_F$.

Similar to the singleton distributed thread vector operators $[\_]_l$, the singleton local thread vector operators $\langle\_\rangle^c_F$ involve an implicit tupling of their operand with a set of foci naming shared services and a natural number less than or equal to k_1.

In the axioms for cyclic distributed interleaving with load balancing introduced below, a binary function app_l ($l \in \mathcal{L}$) from unlocated threads and distributed thread vectors to distributed thread vectors is used which maps each unlocated thread x and distributed thread vector β to the distributed thread vector obtained by appending x to the local thread vector at location l in β and associating the empty set as set of foci and zero as step count with x. The functions app_l are defined in Table 11.

Moreover, a unary function pv on distributed thread vectors is used which

Table 12
Definition of the functions mgc_n and mgc

$mgc_0(x) = \mathsf{T}$	
$mgc_{n+1}(\mathsf{S}) = \mathsf{T}$	
$mgc_{n+1}(\mathsf{D}) = \mathsf{T}$	
$mgc_{n+1}(\mathsf{tau} \circ x) = mgc_n(x)$	
$mgc_{n+1}(x \trianglelefteq f.m \trianglerighteq y) = mgc_n(x) \wedge mgc_n(y)$	if $f = \mathsf{t} \vee (f \in \mathcal{F}_\mathsf{s} \wedge m \in \mathcal{M}_\mathsf{fn})$
$mgc_{n+1}(x \trianglelefteq f.m \trianglerighteq y) = \mathsf{F}$	if $f \neq \mathsf{t} \wedge (f \notin \mathcal{F}_\mathsf{s} \vee m \notin \mathcal{M}_\mathsf{fn})$
$mgc_{n+1}(x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y) = \mathsf{T}$	
$\bigwedge_{n\geq 0} mgc_n(x) = \mathsf{T} \Rightarrow mgc(x) = \mathsf{T}$	
$\bigvee_{n\geq 0} mgc_n(x) = \mathsf{F} \Rightarrow mgc(x) = \mathsf{F}$	

permutes distributed thread vectors cyclicly with implicit migration as outlined above. The unary function pv on distributed thread vectors is defined using a number of auxiliary functions:

- a unary function mgc from unlocated threads to Booleans, mapping each unlocated thread that is migration compatible to T and each unlocated thread that is not migration compatible to F;
- for each $n \in \mathbb{N}$, a unary function nps_n from unlocated threads to Booleans, mapping each unlocated thread to T if the largest number of steps that it will perform before termination, deadlock or explicit migration is at least n, and to F otherwise (nps stands for number of potential steps);
- a unary function $sltv$ from distributed thread vectors to pairs consisting of a location and a local thread vector, mapping each distributed thread vector to the pair consisting of the first among the locations with a shortest local thread vector and the local thread vector at that location;
- a unary function loc from pairs consisting of a location and a local thread vector to locations, mapping each such pair to its first component;
- a unary function tv from pairs consisting of a location and a local thread vector to local thread vectors, mapping each such pair to its second component;
- a unary function imc from distributed thread vectors to Booleans, mapping each distributed thread vector that fulfils the conditions for implicit migration to T and each distributed thread vector that does not fulfil these conditions to F.

The function mgc in turn is defined using, for each $n \in \mathbb{N}$, an auxiliary unary function mgc_n from unlocated threads to Booleans, mapping each unlocated thread whose approximation up to depth n is migration compatible to T and each thread whose approximation up to depth n is not migration compatible to F. The function mgc, as well as the auxiliary functions mgc_n, are defined in Table 12. The function pv, as well as the auxiliary functions nps_n, $sltv$, loc, tv and imc, are defined in Table 13. In the definition of $sltv$, l_0 stands for a fixed but arbitrary location from $\mathcal{L}$.

Table 13
Definition of the functions nps_n, $sltv$, loc, tv, imc, and pv

$nps_0(x) = \mathsf{T}$
$nps_{n+1}(\mathsf{S}) = \mathsf{F}$
$nps_{n+1}(\mathsf{D}) = \mathsf{F}$
$nps_{n+1}(\mathsf{tau} \circ x) = nps_n(x)$
$nps_{n+1}(x \trianglelefteq f.m \trianglerighteq y) = nps_n(x) \vee nps_n(y)$
$nps_{n+1}(x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y) = \mathsf{F}$

$sltv(\langle\,\rangle) = (l_0, \langle\,\rangle)$
$sltv([\alpha]_l) = (l, \alpha)$
$len(\alpha_1) \leq len(\alpha_2) \Rightarrow sltv([\alpha_1]_{l_1} \frown [\alpha_2]_{l_2} \frown \beta) = sltv([\alpha_1]_{l_1} \frown \beta)$
$len(\alpha_1) > len(\alpha_2) \Rightarrow sltv([\alpha_1]_{l_1} \frown [\alpha_2]_{l_2} \frown \beta) = sltv([\alpha_2]_{l_2} \frown \beta)$

$loc((l, \alpha)) = l$
$tv((l, \alpha)) = \alpha$

$imc(\langle\,\rangle) = \mathsf{F}$
$imc([\langle\,\rangle]_l \frown \beta) = \mathsf{F}$
$imc([\langle x\rangle_\emptyset^{k_1} \frown \alpha]_l \frown \beta) = mgc(x) \wedge nps_{k_2}(x) \wedge len(\alpha) > k_3 \wedge 2 \cdot len(tv(sltv(\beta))) < len(\alpha)$
$imc([\langle x\rangle_F^c \frown \alpha]_l \frown \beta) = \mathsf{F}$ if $F \neq \emptyset \vee c \neq k_1$

$pv(\langle\,\rangle) = \langle\,\rangle$
$pv([\langle\,\rangle]_l \frown \beta) = \beta \frown [\langle\,\rangle]_l$
$imc([\langle x\rangle_F^c \frown \alpha]_l \frown \beta) \Rightarrow pv([\langle x\rangle_F^c \frown \alpha]_l \frown \beta) = app_{loc(sltv(\beta))}(x, \beta \frown [\alpha]_l)$
$\neg\ imc([\langle x\rangle_F^c \frown \alpha]_l \frown \beta) \Rightarrow pv([\langle x\rangle_F^c \frown \alpha]_l \frown \beta) = \beta \frown [\alpha \frown \langle x\rangle_F^c]_l$

In the definitions of nps_n, $sltv$, imc and pv, as well as in many subsequent definitions, we use the usual basic functions on Booleans and natural numbers. Moreover, we write $b \Rightarrow e$, where b is a Boolean expression and e is an equation, for the conditional equation $b = \mathsf{T} \Rightarrow e$.

A simple unary function ic on natural numbers is used as well in the axioms for cyclic distributed interleaving with load balancing introduced below. It is defined by $ic(c) = \min(c + 1, k_1)$ for all $c \in \mathbb{N}$.

The axioms for cyclic distributed interleaving with load balancing are given in Table 14. In this table and all subsequent tables with axioms for a distributed interleaving strategy, c stands for an arbitrary natural number less than or equal to k_1. The differences with the axioms of the strategy from the previous section are mainly found in axioms CDIlba6 and CDIlba7a–CDIlba7d. In those axioms, pv is used to achieve the variant of cyclic permutation with implicit migration outlined above.

Guarded recursion can be added to $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$ as it is added to BTA in Section 3.

The distributed interleaving strategy with implicit migration of which the axioms are given in Table 14 does not count failed attempts of a thread to

Table 14
Axioms for cyclic distributed interleaving with load balancing

$\|(\langle\rangle) = \mathsf{S}$	CDIlba1
$\|([\langle\rangle]_{l_1} \frown \ldots \frown [\langle\rangle]_{l_k}) = \mathsf{S}$	CDIlba2
$\|([\langle\rangle]_l \frown \beta) = \|(\beta \frown [\langle\rangle]_l)$	CDIlba3
$\|([\langle \mathsf{S}\rangle_F^c \frown \alpha]_l \frown \beta) = \|(\beta \frown [\alpha]_l)$	CDIlba4
$\|([\langle \mathsf{D}\rangle_F^c \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l))$	CDIlba5
$\|([\langle \mathsf{tau} \circ x\rangle_F^c \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(pv([\langle x\rangle_F^{ic(c)} \frown \alpha]_l \frown \beta))$	CDIlba6
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) =$ $\|(pv([\langle x\rangle_F^{ic(c)} \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.m \trianglerighteq \|(pv([\langle y\rangle_F^{ic(c)} \frown \alpha]_l \frown \beta))$ if $f \notin \mathcal{F}_\mathsf{s} \vee (f \in F \wedge m \notin \mathcal{M}_\mathsf{I})$	CDIlba7a
$\|([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) =$ $\|(\beta \frown [\alpha \frown \langle x\rangle_{F\cup\{f\}}^{ic(c)}]_l) \trianglelefteq l.f.\mathsf{lock} \trianglerighteq \|(pv([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta))$ if $f \in \mathcal{F}_\mathsf{s} \setminus F$	CDIlba7b
$\|([\langle x \trianglelefteq f.\mathsf{unlock} \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) =$ $\|(pv([\langle x\rangle_{F\setminus\{f\}}^{ic(c)} \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.\mathsf{unlock} \trianglerighteq \|(pv([\langle y\rangle_F^{ic(c)} \frown \alpha]_l \frown \beta))$ if $f \in F$	CDIlba7c
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l))$ if $f \in \mathcal{F}_\mathsf{s} \wedge (f \in F \vee m \neq \mathsf{lock}) \wedge (f \in \mathcal{F}_\mathsf{s} \setminus F \vee m = \mathsf{lock})$	CDIlba7d
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(app_n(x, \beta \frown [\alpha]_l))$ if $n \in \mathcal{L} \wedge F = \emptyset$	CDIlba8
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(\beta \frown [\alpha \frown \langle y\rangle_F^c]_l)$ if $n \notin \mathcal{L} \vee F \neq \emptyset$	CDIlba9

acquire a lock on a shared service as steps performed by the thread. Counting the failed attempts is plausible as well. However, it results in looking at past steps and future steps from a different angle because the number of potential failed attempts in the future is generally indefinite.

Because it is assumed that lock is a non-functional method, it is impossible that a thread is implicitly migrated to another location after repeated failed attempts of the thread to acquire a lock on a shared service. At first sight, continuing the attemps at another location after repeated failed attempts appears to make sense, but by doing so the thread runs the risk of hopping around uselessly while attempting to acquire a lock. Therefore, we have not considered the rather counter-intuitive assumption that lock is a functional method.

Migration compatibility is defined such that interaction with services after explicit migration is not taken into account, for that interaction is intended to be with services at another location. Take a thread that cannot be migrated implicitly because of interaction with a target service before termination, deadlock or explicit migration. If that thread would migrate back explicitly before the interaction with a target service instead, then it could be migrated implicitly. Thus, load balancing is feasible and useful because explicit migration is present as an option.

The following might be considered: a thread transformation that inserts load

balancing migration actions to obtain load balancing triggered by actions instead of the distributed interleaving strategy. However, there appear to be no obvious candidate for such a transformation. It can be done, but the state space of the thread may explode.

9 Load Balancing with Implicit Migration Back

In the distributed interleaving strategy with load balancing from Section 8, the current thread is implicitly migrated only if it is migration compatible. Although it is a simple criterion, migration compatibility is undecidable. This makes the distributed interleaving strategy with load balancing from Section 8 unconvincing. In this section, we introduce a variation of that strategy which does without migration compatibility. This results in a theory called $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$.

The approach followed is to preclude the need for migration compatibility on implicit migration of a thread by migrating the thread implicitly back before interaction with a target service or interaction with a shared service using a non-functional method can take place. Because locking is an instance of interaction with a shared service using a non-functional method, implicit migration back does not happen while the thread keeps one or more locks.

Immediately after the current thread has performed an action, implicit migration back may take place. The current thread is implicitly migrated back if the following conditions are fulfilled:

- the most recent migration of the current thread was an implicit migration;
- the next action that the current thread will perform concerns an interaction with a target service or an interaction with a shared service using a non-functional method;
- the location at which the current thread is differs from the location from which its oldest implicit migration that was not followed by an explicit migration has taken place.

If the current thread is implicitly migrated back, it will be migrated to the location from which its oldest implicit migration that was not followed by an explicit migration has taken place.

Whether implicit migration of the current thread takes place, depends as before on the three parameters k_1, k_2 and k_3. The current thread is implicitly migrated if the following conditions are fulfilled:

- the second condition for implicit migration back is not fulfilled;
- the current thread keeps no locks;

- the number of steps that the current thread has performed at its current location is at least k_1;
- the largest number of steps that the current thread will perform before termination, deadlock, explicit migration, interaction with a target service or interaction with a shared service using a non-functional method is at least k_2;
- the length of the local thread vector that contains the current thread, say n, is greater than k_3 and there is another local thread vector whose length times 2 is less than n.

To deal with implicit migration back, we have to enrich distributed thread vectors by replacing each thread in each local thread vector by a quadruple consisting of the thread, the set of all foci naming services on which the thread keeps a lock, the minimum of k_1 and the number of steps that the thread has performed at the location it is, and the location from which the oldest implicit migration of the thread that was not followed by an explicit migration has taken place if it exists and the location at which the thread is otherwise. We will use the term migration-back location to refer to this location.

We mention that, when the interleaving of the threads in a distributed thread vector makes a start, the threads are supposed to keep no lock, to have performed no steps at the location they are, and to have not been implicitly migrated.

$\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$ has the same sorts as $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$. To build terms of the sorts **T**, **LT** and **DTV**, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$ has the same constants and operators as $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$. To build terms of sort **LTV**, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$ has the following constants and operators:

- the *empty local thread vector* constant $\langle\,\rangle : \mathbf{LTV}$;
- for each $F \subseteq \mathcal{F}_{\mathsf{s}}$, $c \in \mathbb{N}$ such that $c \leq k_1$ and $l \in \mathcal{L}$, the unary *singleton local thread vector* operator $\langle \_ \rangle_F^{c,l} : \mathbf{T} \to \mathbf{LTV}$;
- the binary *local thread vector concatenation* operator $\frown : \mathbf{LTV} \times \mathbf{LTV} \to \mathbf{LTV}$.

That is, the operator $\langle \_ \rangle_F^c$ is replaced by the operators $\langle \_ \rangle_F^{c,l}$.

Similar to the singleton distributed thread vector operators $[\_]_l$, the singleton local thread vector operators $\langle \_ \rangle_F^{c,l}$ involve an implicit tupling of their operand with a set of foci naming shared services, a natural number less than or equal to k_1, and a location.

In the axioms for cyclic distributed interleaving with load balancing and implicit migration back introduced below, a binary function $app_{l,o}$ $(l, o \in \mathcal{L})$ from unlocated threads and distributed thread vectors to distributed thread vectors is used which maps each unlocated thread x and distributed thread vector β to the distributed thread vector obtained by appending x to the local thread

Table 15
Definition of the functions $app_{l,o}$

$app_{l,o}(x, \langle\rangle) = \langle\rangle$
$app_{l,o}(x, [\alpha]_{l'} \frown \beta) = [\alpha \frown \langle x \rangle_{\emptyset}^{0,o}]_l \frown \beta$ if $l = l'$
$app_{l,o}(x, [\alpha]_{l'} \frown \beta) = [\alpha]_{l'} \frown app_{l,o}(x, \beta)$ if $l \neq l'$

vector at location l in β and associating the empty set as set of foci, zero as step count and o as migration-back location with x. The functions $app_{l,o}$ are defined in Table 15.

Moreover, a unary function pv on distributed thread vectors is used which permutes distributed thread vectors cyclicly with implicit migration and implicit migration back as outlined above. The unary function pv on distributed thread vectors is defined using auxiliary functions nps_n, $sltv$, loc, tv and imc which are essentially the same as the ones used before in the case of the strategy from Section 8, with the exception of the functions nps_n and imc. The functions nps_n and imc become slightly different:

- for each $n \in \mathbb{N}$, nps_n maps each unlocated thread to T if the largest number of steps that it will perform before termination, deadlock, explicit migration, interaction with a target service or interaction with a shared service using a non-functional method is at least n, and to F otherwise;
- imc maps each distributed thread vector that fulfils the adapted conditions for implicit migration to T and each distributed thread vector that does not fulfil these conditions to F.

We also use the following auxiliary functions:

- a unary function $imgc$ from unlocated threads to Booleans, mapping each unlocated thread that does not fulfil the second condition for implicit migration back to T and each unlocated thread that fulfils the second condition for implicit migration back to F ($imgc$ stands for initially migration compatible);
- a unary function $imbc$ from distributed thread vectors to Booleans, mapping each distributed thread vector that fulfils the conditions for implicit migration back to T and each distributed thread vector that does not fulfil the conditions for implicit migration back to F.

The function pv, as well as the auxiliary functions nps_n, $sltv$, loc, tv, $imgc$, $imbc$ and imc, are defined in Table 16. In the definition of $sltv$, l_0 stands for a fixed but arbitrary location from $\mathcal{L}$.

The axioms for cyclic distributed interleaving with load balancing and implicit migration back are given in Table 17. In this table, o stands for an arbitrary location from $\mathcal{L}$. There seems to be no essential difference with the axioms of the strategy from the previous section. However, the defining equations of

Table 16
Definition of the functions nps_n, $sltv$, loc, tv, $imgc$, $imbc$, imc, and pv

$nps_0(x) = \mathsf{T}$
$nps_{n+1}(\mathsf{S}) = \mathsf{F}$
$nps_{n+1}(\mathsf{D}) = \mathsf{F}$
$nps_{n+1}(\mathsf{tau} \circ x) = nps_n(x)$
$nps_{n+1}(x \trianglelefteq f.m \trianglerighteq y) = nps_n(x) \vee nps_n(y)$ if $f = \mathsf{t} \vee (f \in \mathcal{F}_\mathsf{s} \wedge m \in \mathcal{M}_\mathsf{fn})$
$nps_{n+1}(x \trianglelefteq f.m \trianglerighteq y) = \mathsf{F}$ if $f \neq \mathsf{t} \wedge (f \notin \mathcal{F}_\mathsf{s} \vee m \notin \mathcal{M}_\mathsf{fn})$
$nps_{n+1}(x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y) = \mathsf{F}$

$sltv(\langle\,\rangle) = (l_0, \langle\,\rangle)$
$sltv([\alpha]_l) = (l, \alpha)$
$len(\alpha_1) \leq len(\alpha_2) \Rightarrow sltv([\alpha_1]_{l_1} \frown [\alpha_2]_{l_2} \frown \beta) = sltv([\alpha_1]_{l_1} \frown \beta)$
$len(\alpha_1) > len(\alpha_2) \Rightarrow sltv([\alpha_1]_{l_1} \frown [\alpha_2]_{l_2} \frown \beta) = sltv([\alpha_2]_{l_2} \frown \beta)$

$loc((l, \alpha)) = l$
$tv((l, \alpha)) = \alpha$

$imgc(\mathsf{S}) = \mathsf{T}$
$imgc(\mathsf{D}) = \mathsf{T}$
$imgc(\mathsf{tau} \circ x) = \mathsf{T}$
$imgc(x \trianglelefteq f.m \trianglerighteq y) = \mathsf{T}$ if $f = \mathsf{t} \vee (f \in \mathcal{F}_\mathsf{s} \wedge m \in \mathcal{M}_\mathsf{fn})$
$imgc(x \trianglelefteq f.m \trianglerighteq y) = \mathsf{F}$ if $f \neq \mathsf{t} \wedge (f \notin \mathcal{F}_\mathsf{s} \vee m \notin \mathcal{M}_\mathsf{fn})$
$imgc(x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y) = \mathsf{T}$

$imbc(\langle\,\rangle) = \mathsf{F}$
$imbc([\langle\,\rangle]_l \frown \beta) = \mathsf{F}$
$imbc([\langle x \rangle_F^{c,l} \frown \alpha]_l \frown \beta) = \mathsf{F}$
$imbc([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) = \neg\ imgc(x)$ if $l \neq o$

$imc(\langle\,\rangle) = \mathsf{F}$
$imc([\langle\,\rangle]_l \frown \beta) = \mathsf{F}$
$imc([\langle x \rangle_\emptyset^{k_1,o} \frown \alpha]_l \frown \beta) = imgc(x) \wedge nps_{k_2}(x) \wedge len(\alpha) > k_3 \wedge 2 \cdot len(tv(sltv(\beta))) < len(\alpha)$
$imc([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) = \mathsf{F}$ if $F \neq \emptyset \vee c \neq k_1$

$pv(\langle\,\rangle) = \langle\,\rangle$
$pv([\langle\,\rangle]_l \frown \beta) = \beta \frown [\langle\,\rangle]_l$
$imbc([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) \Rightarrow pv([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) = app_{o,o}(x, \beta \frown [\alpha]_l)$
$imc([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) \Rightarrow pv([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) = app_{loc(sltv(\beta)),o}(x, \beta \frown [\alpha]_l)$
$\neg\ imbc([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) \wedge \neg\ imc([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) \Rightarrow$
$\quad pv([\langle x \rangle_F^{c,o} \frown \alpha]_l \frown \beta) = \beta \frown [\alpha \frown \langle x \rangle_F^{c,o}]_l$

the function pv used here (Table 16) show that in this case pv deals with the variant of cyclic permutation with implicit migration and implicit migration back outlined above.

Guarded recursion can be added to $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$ as it is added to BTA in Section 3.

Like in the strategy with load balancing from Section 8, it is impossible that a thread is implicitly migrated to another location after repeated failed attempts

Table 17
Axioms for cyclic distributed interleaving with load balancing and implicit migration back

$\|(\langle\rangle) = \mathsf{S}$	CDIlbb1
$\|([\langle\rangle]_{l_1} \frown \ldots \frown [\langle\rangle]_{l_k}) = \mathsf{S}$	CDIlbb2
$\|([\langle\rangle]_l \frown \beta) = \|(\beta \frown [\langle\rangle]_l)$	CDIlbb3
$\|([\langle \mathsf{S}\rangle_F^{c,o} \frown \alpha]_l \frown \beta) = \|(\beta \frown [\alpha]_l)$	CDIlbb4
$\|([\langle \mathsf{D}\rangle_F^{c,o} \frown \alpha]_l \frown \beta) = \mathsf{S_D}(\|(\beta \frown [\alpha]_l))$	CDIlbb5
$\|([\langle \mathsf{tau} \circ x\rangle_F^{c,o} \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(pv([\langle x\rangle_F^{ic(c),o} \frown \alpha]_l \frown \beta))$	CDIlbb6
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta) =$ $\|(pv([\langle x\rangle_F^{ic(c),o} \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.m \trianglerighteq \|(pv([\langle y\rangle_F^{ic(c),o} \frown \alpha]_l \frown \beta))$ if $f \notin \mathcal{F}_\mathsf{s} \vee (f \in F \wedge m \notin \mathcal{M}_\mathsf{l})$	CDIlbb7a
$\|([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta) =$ $\|(\beta \frown [\alpha \frown \langle x\rangle_{F\cup\{f\}}^{ic(c),o}]_l) \trianglelefteq l.f.\mathsf{lock} \trianglerighteq \|(pv([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta))$ if $f \in \mathcal{F}_\mathsf{s} \setminus F$	CDIlbb7b
$\|([\langle x \trianglelefteq f.\mathsf{unlock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta) =$ $\|(pv([\langle x\rangle_{F\setminus\{f\}}^{ic(c),o} \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.\mathsf{unlock} \trianglerighteq \|(pv([\langle y\rangle_F^{ic(c),o} \frown \alpha]_l \frown \beta))$ if $f \in F$	CDIlbb7c
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta) = \mathsf{S_D}(\|(\beta \frown [\alpha]_l))$ if $f \in \mathcal{F}_\mathsf{s} \wedge (f \in F \vee m \neq \mathsf{lock}) \wedge (f \in \mathcal{F}_\mathsf{s} \setminus F \vee m = \mathsf{lock})$	CDIlbb7d
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(app_{n,n}(x, \beta \frown [\alpha]_l))$ if $n \in \mathcal{L} \wedge F = \emptyset$	CDIlbb8
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(\beta \frown [\alpha \frown \langle y\rangle_F^{c,o}]_l)$ if $n \notin \mathcal{L} \vee F \neq \emptyset$	CDIlbb9

to acquire a lock on a shared service. Using implicit migration back, continuing the attemps at another location after repeated failed attempts becomes rather troublesome: it may undo a necessary implicit migration back that has taken place before the first attempt to acquire the lock.

10 Handling Nodes with Different Processing Speeds

In Sections 8 and 9, load balancing is implicitly based on the assumption that the speed with which actions are processed at the different locations are the same. In this section, we introduce a variation of the distributed interleaving strategy with load balancing and implicit migration back from Section 9 where load balancing is based on the assumption that the speed with which actions are processed at the different locations is not necessarily the same. This results in a theory called $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbs}}$.

Like before, assume that time is divided into slices of equal length. Suppose that during each slice, each local thread vector gets a turn to perform a number of actions of one thread where that number depends upon the location of the local thread vector. This means that the speed with which actions are processed at the different locations is not necessarily the same. Clearly, the

execution speed of the threads in a local thread vector depends upon the speed with which actions are processed. This must be dealt with in equalizing the execution speed of the threads in the different local thread vectors of a distributed thread vector.

It is assumed that there is an *action processing capacity* function *apc* from locations to natural numbers. This function is regarded to give for each location the number of actions that are performed by the local thread vector at that location during one turn.

To deal in load balancing with the situation that actions are processed with different speeds at different locations, we consider the length of every local thread vector in proportion to the number of actions that are performed by the local thread vector during one turn. Moreover, we add the following condition to the conditions for implicit migration:

- the local thread vector at the current location is not in the middle of a turn to perform a number of actions.

The purpose of this additional condition is to take care that processing capacity is not wasted.

We have to enrich distributed thread vectors once more. The new distributed thread vectors are sequences of triples, one for each location, consisting of a location, the local thread vector at that location, and the number of actions that has been performed during one turn of that local thread vector after it has performed the following action.

We repeat that, when the interleaving of the threads in a distributed thread vector makes a start, the threads are supposed to keep no lock, to have performed no steps at the location they are, and to have not been implicitly migrated.

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

- the *empty distributed thread vector* constant $\langle\,\rangle : \mathbf{DTV}$;
- for each $l \in \mathcal{L}$ and $i \in \mathbb{N}$ such that $i \leq \mathit{apc}(l)$, the unary *singleton distributed thread vector* operator $[\_]_l^i : \mathbf{LTV} \to \mathbf{DTV}$;
- the binary *distributed thread vector concatenation* operator $\frown : \mathbf{DTV} \times \mathbf{DTV} \to \mathbf{DTV}$.

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

The singleton distributed thread vector operators $[\_]_l^i$ involve an implicit tu-

Table 18
Definition of the functions $app_{l,o}$

$app_{l,o}(x, \langle\rangle) = \langle\rangle$
$app_{l,o}(x, [\alpha]^i_{l'} \frown \beta) = [\alpha \frown \langle x \rangle^{0,o}_{\emptyset}]^i_l \frown \beta$ if $l = l'$
$app_{l,o}(x, [\alpha]^i_{l'} \frown \beta) = [\alpha]^i_{l'} \frown app_{l,o}(x, \beta)$ if $l \neq l'$

pling of their operand with a location and a natural number.

In the axioms for cyclic distributed interleaving with load balancing and implicit migration back in case of different processing speeds introduced below, a binary function $app_{l,o}$ $(l, o \in \mathcal{L})$ from unlocated threads and distributed thread vectors to distributed thread vectors is used which maps each unlocated thread x and distributed thread vector β to the distributed thread vector obtained by appending x to the local thread vector at location l in β and associating the empty set as set of foci, zero as step count and o as migration-back location with x. The functions $app_{l,o}$ are defined in Table 18.

Moreover, a unary function pv on distributed thread vectors is used which permutes distributed thread vectors cyclicly with implicit migration and implicit migration back, dealing with the situation that actions are processed with different speeds at different locations as outlined above. The unary function pv on distributed thread vectors is defined using auxiliary functions nps_n, $sltv$, loc, tv, $imgc$, $imbc$ and imc which are the same or essentially the same as the ones used before in the case of the strategy from Section 9, with the exception of the functions $sltv$ and imc. The functions $sltv$ and imc become slightly different:

- $sltv$ maps each distributed thread vector to the pair consisting of: (i) the first among the locations with a shortest local thread vector in proportion to the number of actions that are performed by the local thread vector at that location during one turn; (ii) the local thread vector at that location;
- imc maps each distributed thread vector that fulfils the adapted conditions for implicit migration to T and each distributed thread vector that does not fulfil these conditions to F.

The functions nps_n, loc, tv and $imgc$ are the ones defined before in Table 16. The function pv, as well as the auxiliary functions $sltv$, $imbc$ and imc, are defined in Table 19. In the definition of $sltv$, l_0 stands for a fixed but arbitrary location from $\mathcal{L}$.

The axioms for cyclic distributed interleaving with load balancing and implicit migration back in case of different processing speeds are given in Table 20. In this table, o stands for an arbitrary location from $\mathcal{L}$ and i stand for an arbitrary natural number such that $i \leq apc(l)$. Most axioms of the strategy from the previous section are now replaced by two axioms to make distinction between the cases $i \neq apc(l)$ and $i = apc(l)$. This distinction must be made because

Table 19
Definition of the functions *sltv*, *imbc*, *imc*, and *pv*

$$sltv(\langle\,\rangle) = (l_0, \langle\,\rangle)$$
$$sltv([\alpha]^i_l) = (l, \alpha)$$
$$len(\alpha_1) \cdot apc(l_2) \leq len(\alpha_2) \cdot apc(l_1) \Rightarrow sltv([\alpha_1]^{i_1}_{l_1} \frown [\alpha_2]^{i_2}_{l_2} \frown \beta) = sltv([\alpha_1]^{i_1}_{l_1} \frown \beta)$$
$$len(\alpha_1) \cdot apc(l_2) > len(\alpha_2) \cdot apc(l_1) \Rightarrow sltv([\alpha_1]^{i_1}_{l_1} \frown [\alpha_2]^{i_2}_{l_2} \frown \beta) = sltv([\alpha_2]^{i_2}_{l_2} \frown \beta)$$

$$imbc(\langle\,\rangle) = \mathsf{F}$$
$$imbc([\langle\,\rangle]^i_l \frown \beta) = \mathsf{F}$$
$$imbc([\langle x\rangle^{c,l}_F \frown \alpha]^i_l \frown \beta) = \mathsf{F}$$
$$imbc([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) = \neg\ imgc(x) \qquad \text{if } l \neq o$$

$$imc(\langle\,\rangle) = \mathsf{F}$$
$$imc([\langle\,\rangle]^i_l \frown \beta) = \mathsf{F}$$
$$imc([\langle x\rangle^{k_1,o}_\emptyset \frown \alpha]^1_l \frown \beta) =$$
$$\quad imgc(x) \wedge nps_{k_2}(x) \wedge$$
$$\quad len(\alpha) > k_3 \cdot apc(l) \wedge 2 \cdot len(tv(sltv(\beta))) \cdot apc(l) < len(\alpha) \cdot apc(loc(sltv(\beta)))$$
$$imc([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) = \mathsf{F} \qquad \text{if } F \neq \emptyset \vee c \neq k_1 \vee i \neq 1$$

$$pv(\langle\,\rangle) = \langle\,\rangle$$
$$pv([\langle\,\rangle]^i_l \frown \beta) = \beta \frown [\langle\,\rangle]^1_l$$
$$imbc([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) \Rightarrow pv([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) = app_{o,o}(x, \beta \frown [\alpha]^1_l)$$
$$imc([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) \Rightarrow pv([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) = app_{loc(sltv(\beta)),o}(x, \beta \frown [\alpha]^1_l)$$
$$\neg\ imbc([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) \wedge \neg\ imc([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) \Rightarrow$$
$$\quad pv([\langle x\rangle^{c,o}_F \frown \alpha]^i_l \frown \beta) = \beta \frown [\alpha \frown \langle x\rangle^{c,o}_F]^i_l$$

permutation should only take place after the current thread has performed $apc(l)$ actions.

Guarded recursion can be added to $\mathrm{TA}^{\mathrm{lbs}}_{\mathrm{dsi}}$ as it is added to BTA in Section 3.

In the strategy introduced here, different processing speeds at different locations are taken into account by giving the local thread vectors at different locations different numbers of actions of one thread to perform during one turn. Alternatively, they can be taken into account by giving the local thread vectors at different locations one action of different numbers of threads to perform during one turn.

11 Distributed Strategic Interleaving with Capability Searching

In preceding sections, it was assumed that the same services are available at each location. In the case of execution architectures where this assumption holds, distributed interleaving strategies with implicit migration of threads to achieve load balancing are plausible. In the case of execution architectures where this assumption does not hold, distributed interleaving strategies with

Table 20
Axioms for cyclic distributed interleaving with load balancing in case of different processing speeds

$\|(\langle\rangle) = \mathsf{S}$	CDIlbs1
$\|([\langle\rangle]_{l_1}^{i_1} \frown \ldots \frown [\langle\rangle]_{l_k}^{i_k}) = \mathsf{S}$	CDIlbs2
$\|([\langle\rangle]_l^i \frown \beta) = \|(\beta \frown [\langle\rangle]_l^1)$	CDIlbs3
$\|([\langle \mathsf{S}\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = \|(\beta \frown [\alpha]_l^1)$	CDIlbs4
$\|([\langle \mathsf{D}\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l^1))$	CDIlbs5
$\|([\langle \mathsf{tau} \circ x\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = l.\mathsf{tau} \circ \|([\langle x\rangle_F^{ic(c),o} \frown \alpha]_l^{i+1} \frown \beta)$ if $i \neq apc(l)$	CDIlbs6a
$\|([\langle \mathsf{tau} \circ x\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = l.\mathsf{tau} \circ \|(pv([\langle x\rangle_F^{ic(c),o} \frown \alpha]_l^1 \frown \beta))$ if $i = apc(l)$	CDIlbs6b
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) =$ $\|([\langle x\rangle_F^{ic(c),o} \frown \alpha]_l^{i+1} \frown \beta) \trianglelefteq l.f.m \trianglerighteq \|([\langle y\rangle_F^{ic(c),o} \frown \alpha]_l^{i+1} \frown \beta)$ if $(f \notin \mathcal{F}_\mathsf{s} \vee (f \in F \wedge m \notin \mathcal{M}_\mathsf{I})) \wedge i \neq apc(l)$	CDIlbs7aa
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) =$ $\|(pv([\langle x\rangle_F^{ic(c),o} \frown \alpha]_l^1 \frown \beta)) \trianglelefteq l.f.m \trianglerighteq \|(pv([\langle y\rangle_F^{ic(c),o} \frown \alpha]_l^1 \frown \beta))$ if $(f \notin \mathcal{F}_\mathsf{s} \vee (f \in F \wedge m \notin \mathcal{M}_\mathsf{I})) \wedge i = apc(l)$	CDIlbs7ab
$\|([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) =$ $\|([\langle x\rangle_{F\cup\{f\}}^{ic(c),o} \frown \alpha]_l^{i+1} \frown \beta) \trianglelefteq l.f.\mathsf{lock} \trianglerighteq \|([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^{i+1} \frown \beta))$ if $f \in \mathcal{F}_\mathsf{s} \setminus F \wedge i \neq apc(l)$	CDIlbs7ba
$\|([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) =$ $\|(\beta \frown [\alpha \frown \langle x\rangle_{F\cup\{f\}}^{ic(c),o}]_l^1) \trianglelefteq l.f.\mathsf{lock} \trianglerighteq \|(pv([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^1 \frown \beta))$ if $f \in \mathcal{F}_\mathsf{s} \setminus F \wedge i = apc(l)$	CDIlbs7bb
$\|([\langle x \trianglelefteq f.\mathsf{unlock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) =$ $\|([\langle x\rangle_{F\setminus\{f\}}^{ic(c),o} \frown \alpha]_l^{i+1} \frown \beta) \trianglelefteq l.f.\mathsf{unlock} \trianglerighteq \|([\langle y\rangle_F^{ic(c),o} \frown \alpha]_l^{i+1} \frown \beta)$ if $f \in F \wedge i \neq apc(l)$	CDIlbs7ca
$\|([\langle x \trianglelefteq f.\mathsf{unlock} \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) =$ $\|(pv([\langle x\rangle_{F\setminus\{f\}}^{ic(c),o} \frown \alpha]_l^1 \frown \beta)) \trianglelefteq l.f.\mathsf{unlock} \trianglerighteq \|(pv([\langle y\rangle_F^{ic(c),o} \frown \alpha]_l^1 \frown \beta))$ if $f \in F \wedge i = apc(l)$	CDIlbs7cb
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l^1))$ if $f \in \mathcal{F}_\mathsf{s} \wedge (f \in F \vee m \neq \mathsf{lock}) \wedge (f \in \mathcal{F}_\mathsf{s} \setminus F \vee m = \mathsf{lock})$	CDIlbs7d
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = l.\mathsf{tau} \circ \|(app_{n,n}(x, \beta \frown [\alpha]_l^1))$ if $n \in \mathcal{L} \wedge F = \emptyset$	CDIlbs8
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = l.\mathsf{tau} \circ \|([\langle y\rangle_F^{c,o} \frown \alpha]_l^{i+1} \frown \beta)$ if $n \notin \mathcal{L} \vee F \neq \emptyset \wedge i \neq apc(l)$	CDIlbs9a
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F^{c,o} \frown \alpha]_l^i \frown \beta) = l.\mathsf{tau} \circ \|(\beta \frown [\alpha \frown \langle y\rangle_F^{c,o}]_l^1)$ if $n \notin \mathcal{L} \vee F \neq \emptyset \wedge i = apc(l)$	CDIlbs9b

implicit migration of threads to achieve availability of services needed by the thread are plausible. We say that such distributed interleaving strategies take care of capability searching. In this section, we introduce a variation of the distributed interleaving strategy from Section 7 with capability searching. This results in a theory called $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{cs}}$.

The distributed interleaving strategy with capability searching has one pa-

Table 21
Definition of the functions app_L

$app_L(x, \langle\rangle) = \langle\rangle$
$app_L(x, [\alpha]_l \frown \beta) = [\alpha \frown \langle x \rangle_\emptyset]_l \frown \beta$ if $l \in L$
$app_L(x, [\alpha]_l \frown \beta) = [\alpha]_l \frown app_L(x, \beta)$ if $l \notin L$

rameter: k. This parameter is greater then 0. Immediately after the current thread has performed an action, implicit migration of that thread to another location may take place. Whether migration really takes place, depends on the services present at the current location. The current thread is implicitly migrated if the following conditions are fulfilled:

- the action that the current thread will perform next has to be processed by a service that is not present at the current location;
- the current thread keeps no locks.

If these conditions are fulfilled, then the current thread will be migrated to the first among the locations where the least number of steps that its approximation up to depth k must perform to encounter an action that has to be processed by a service that is not present at that location is maximal. If the first condition is fulfilled, but the second condition is not fulfilled, then the current thread will deadlock on its next turn to perform an action.

Notice that the distributed interleaving strategy with capability searching does not fit in with the view that interaction of threads with services may in whole or in part be bound to services at the location the threads are.

It is assumed that there is a function *foci* from locations to sets of foci. This function is regarded to give for each location the set of all foci naming services present at that location. It is assumed that, for all $f \in \mathcal{F} \setminus \{\mathsf{t}\}$, there exists an $l \in \mathcal{L}$ such that $f \in foci(l)$. Moreover, it is assumed that, for all $f \in \mathcal{F} \setminus (\mathcal{F}_\mathsf{s} \cup \{\mathsf{t}\})$, there exists a unique $l \in \mathcal{L}$ such that $f \in foci(l)$. The last two assumptions mean that each service is available somewhere and each target service is available at no more than one location. We believe that the healtiness of the distributed interleaving strategy introduced here is questionable without these assumptions.

$\mathrm{TA}^{\mathrm{cs}}_{\mathrm{dsi}}$ has the sorts, constants and operators of $\mathrm{TA}^{\mathrm{lck}}_{\mathrm{dsi}}$.

In the axioms for cyclic distributed interleaving with capability searching introduced below, a binary function app_L ($L \subseteq \mathcal{L}$) from unlocated threads and distributed thread vectors to distributed thread vectors is used which maps each unlocated thread x and distributed thread vector β to the distributed thread vector obtained by appending x to the first local thread vector at a location from L in β and associating the empty set as set of foci with x. The functions app_L are defined in Table 21.

Table 22
Definition of the functions $locs'_n$, $locs_n$, and pv

$$
\begin{array}{l}
locs'_0(x) = \mathcal{L} \\
locs'_{n+1}(\mathsf{S}) = \mathcal{L} \\
locs'_{n+1}(\mathsf{D}) = \mathcal{L} \\
locs'_{n+1}(\mathsf{tau} \circ x) = locs'_n(x) \\
locs'_{n+1}(x \trianglelefteq f.m \trianglerighteq y) = locs'_n(x) \cap \{l \mid f \in foci(l)\} \cap locs'_n(y) \\
locs'_{n+1}(x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y) = locs'_n(y) \\
\\
locs_0(x) = \emptyset \\
locs'_{n+1}(x) \neq \emptyset \Rightarrow locs_{n+1}(x) = locs'_{n+1}(x) \\
locs'_{n+1}(x) = \emptyset \Rightarrow locs_{n+1}(x) = locs_n(x) \\
\\
imc(\langle\,\rangle) = \mathsf{F} \\
imc([\langle\,\rangle]_l \frown \beta) = \mathsf{F} \\
imc([\langle \mathsf{S} \rangle_F \frown \alpha]_l \frown \beta) = \mathsf{F} \\
imc([\langle \mathsf{D} \rangle_F \frown \alpha]_l \frown \beta) = \mathsf{F} \\
imc([\langle \mathsf{tau} \circ x \rangle_F \frown \alpha]_l \frown \beta) = \mathsf{F} \\
imc([\langle x \trianglelefteq f.m \trianglerighteq y \rangle_F \frown \alpha]_l \frown \beta) = \mathsf{F} \qquad \text{if } f \in foci(l) \vee F \neq \emptyset \\
imc([\langle x \trianglelefteq f.m \trianglerighteq y \rangle_F \frown \alpha]_l \frown \beta) = \mathsf{T} \qquad \text{if } f \notin foci(l) \wedge F = \emptyset \\
imc([\langle x \trianglelefteq \mathsf{mg}(n') \trianglerighteq y \rangle_F \frown \alpha]_l \frown \beta) = \mathsf{F} \\
\\
pv(\langle\,\rangle) = \langle\,\rangle \\
pv([\langle\,\rangle]_l \frown \beta) = \beta \frown [\langle\,\rangle]_l \\
imc([\langle x \rangle_F \frown \alpha]_l \frown \beta) \Rightarrow pv([\langle x \rangle_F \frown \alpha]_l \frown \beta) = app_{locs_k(x)}(x, \beta \frown [\alpha]_l) \\
\neg\ imc([\langle x \rangle_F \frown \alpha]_l \frown \beta) \Rightarrow pv([\langle x \rangle_F \frown \alpha]_l \frown \beta) = \beta \frown [\alpha \frown \langle x \rangle_F]_l
\end{array}
$$

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

- for each $n \in \mathbb{N}$, a unary function $locs'_n$ from unlocated threads to sets of locations, mapping each unlocated thread to the set of all locations at which all services are present that may be needed to process the actions performed by its approximation up to depth n;
- for each $n \in \mathbb{N}$, a unary function $locs_n$ from unlocated threads to sets of locations, mapping each unlocated thread to the set of all locations where the least number of steps that its approximation up to depth n must perform to encounter an action that has to be processed by a service that is not present at the location is maximal;
- a unary function imc from distributed thread vectors to Booleans, mapping each distributed thread vector that fulfils the adapted conditions for implicit migration to T and each distributed thread vector that does not fulfil these conditions to F.

The function pv, as well as the auxiliary functions $locs'_n$, $locs_n$ and imc, are defined in Table 22.

Table 23
Axioms for cyclic distributed interleaving with capability searching

$\|(\langle\rangle) = \mathsf{S}$	CDIcs1
$\|([\langle\rangle]_{l_1} \frown \ldots \frown [\langle\rangle]_{l_k}) = \mathsf{S}$	CDIcs2
$\|([\langle\rangle]_l \frown \beta) = \|(\beta \frown [\langle\rangle]_l)$	CDIcs3
$\|([\langle \mathsf{S}\rangle_F \frown \alpha]_l \frown \beta) = \|(\beta \frown [\alpha]_l)$	CDIcs4
$\|([\langle \mathsf{D}\rangle_F \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l))$	CDIcs5
$\|([\langle \mathsf{tau} \circ x\rangle_F \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(pv([\langle x\rangle_F \frown \alpha]_l \frown \beta))$	CDIcs6
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) =$	
$\quad \|(pv([\langle x\rangle_F \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.m \trianglerighteq \|(pv([\langle y\rangle_F \frown \alpha]_l \frown \beta))$	
$\quad \mathsf{if}\ f \in \mathit{foci}(l) \wedge (f \notin \mathcal{F}_\mathsf{s} \vee (f \in F \wedge m \notin \mathcal{M}_\mathsf{l}))$	CDIcs7a
$\|([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) =$	
$\quad \|(pv([\langle x\rangle_{F\cup\{f\}} \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.\mathsf{lock} \trianglerighteq \|(pv([\langle x \trianglelefteq f.\mathsf{lock} \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta))$	
$\quad \mathsf{if}\ f \in \mathit{foci}(l) \wedge f \in \mathcal{F}_\mathsf{s} \setminus F$	CDIcs7b
$\|([\langle x \trianglelefteq f.\mathsf{unlock} \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) =$	
$\quad \|(pv([\langle x\rangle_{F\setminus\{f\}} \frown \alpha]_l \frown \beta)) \trianglelefteq l.f.\mathsf{unlock} \trianglerighteq \|(pv([\langle y\rangle_F \frown \alpha]_l \frown \beta))$	
$\quad \mathsf{if}\ f \in \mathit{foci}(l) \wedge f \in F$	CDIcs7c
$\|([\langle x \trianglelefteq f.m \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) = \mathsf{S}_\mathsf{D}(\|(\beta \frown [\alpha]_l))$	
$\quad \mathsf{if}\ f \notin \mathit{foci}(l) \vee (f \in \mathcal{F}_\mathsf{s} \wedge (f \in F \vee m \neq \mathsf{lock}) \wedge (f \in \mathcal{F}_\mathsf{s} \setminus F \vee m = \mathsf{lock}))$	CDIcs7d
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(\mathit{app}_{\{n\}}(x, \beta \frown [\alpha]_l))\ \ \mathsf{if}\ n \in \mathcal{L} \wedge F = \emptyset$	CDIcs8
$\|([\langle x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y\rangle_F \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(pv([\langle y\rangle_F \frown \alpha]_l \frown \beta))\ \ \mathsf{if}\ n \notin \mathcal{L} \vee F \neq \emptyset$	CDIcs9

The axioms for cyclic distributed interleaving with capability searching are given in Table 23. Like with the axioms for the strategy from Section 8, the differences with the axioms for the strategy from Section 7 are mainly found in axioms CDIcs6 and CDIcs7a–CDIcs7d. In those axioms, pv is used to achieve the variant of cyclic permutation with capability searching outlined above.

Guarded recursion can be added to $\mathrm{TA}^{\mathrm{cs}}_{\mathrm{dsi}}$ as it is added to BTA in Section 3.

12 Interaction of Threads with Services

A thread may perform certain actions only for the sake of getting reply values returned by para-target services and that way having itself affected by that service. In this section, we introduce thread-service composition, which allows for threads to be affected by para-target services in this way.

We introduce yet another sort: the sort **PTS** of *para-target services*. However, we will not introduce constants and operators to build terms of this sort. **PTS** is a parameter of theories with thread-service composition. **PTS** is considered to stand for the set of all para-target services. It is assumed that each para-target service can be represented by a function $H : \mathcal{M}^+ \to \{\mathsf{T}, \mathsf{F}, \mathsf{B}\}$ with the property that $H(\gamma) = \mathsf{B} \Rightarrow H(\gamma \frown \langle m\rangle) = \mathsf{B}$ for all $\gamma \in \mathcal{M}^+$ and $m \in \mathcal{M}$. This function is called the *reply* function of the para-target service. A para-target

service fails to produce a reply in the case where it does not accept a request to process a command (see also the next paragraph). This case is modelled by its reply function producing B instead of T or F. Here, B stands for blocked. Given a reply function H and a method $m \in \mathcal{M}$, the *derived* reply function of H after processing m, written $\frac{\partial}{\partial m}H$, is defined by $\frac{\partial}{\partial m}H(\gamma) = H(\langle m\rangle \frown \gamma)$.

The connection between a reply function H and the para-target 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 accepted by the service.

Henceforth, we will identify a reply function with the para-target service represented by it.

For each $l \in \mathcal{L}$ and $f \in \mathcal{F}_\mathsf{s}$, we introduce the binary *thread-shared-service composition* operator $\_ /_{l.f} \_ : \mathbf{LT} \times \mathbf{PTS} \to \mathbf{LT}$. Intuitively, $p /_{l.f} H$ is the thread that results from processing all located actions performed by thread p that are of the form $l.f.m$ by shared service H at location l. When a located action performed by thread p is processed by shared service H at location l, it is turned into the action $l.\mathsf{tau}$ and postconditional composition is removed in favour of located action prefixing on the basis of the reply value produced.

Moreover, we introduce the binary *thread-private-service composition* operator $\_ /_\mathsf{t} \_ : \mathbf{T} \times \mathbf{PTS} \to \mathbf{T}$. Intuitively, $p /_\mathsf{t} H$ is the thread that results from processing all unlocated actions performed by thread p that are of the form $\mathsf{t}.m$ by the private service H of thread p. When an unlocated action performed by thread p is processed by the private service H of p, it is turned into the action tau and postconditional composition is removed in favour of unlocated action prefixing on the basis of the reply value produced.

The axioms for the thread-shared-service composition operators are given in Table 24. In this table, l and l' stand for arbitrary locations from $\mathcal{L}$, f stands for an arbitrary focus from $\mathcal{F}_\mathsf{s}$, f' stands for an arbitrary focus from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$, and n stands for an arbitrary natural number. Axiom TSSC3 shows that the action $l.\mathsf{tau}$ is always accepted. Axioms TSSC5 and TSSC6 make it clear that $l.\mathsf{tau}$ arises as the residue of processing actions at location l. Therefore, $l.\mathsf{tau}$ is not connected to a particular focus and is always accepted. Axiom TSSC7 expresses that deadlock takes place when an action is not accepted.

The axioms for the thread-private-service composition operator are given in

Table 24
Axioms for thread-shared-service composition

$\mathsf{S} \mathbin{/_{l.f}} H = \mathsf{S}$		TSSC1
$\mathsf{D} \mathbin{/_{l.f}} H = \mathsf{D}$		TSSC2
$(l'.\mathsf{tau} \circ u) \mathbin{/_{l.f}} H = l'.\mathsf{tau} \circ (u \mathbin{/_{l.f}} H)$		TSSC3
$(u \trianglelefteq l'.f'.m \trianglerighteq v) \mathbin{/_{l.f}} H = (u \mathbin{/_{l.f}} H) \trianglelefteq l'.f'.m \trianglerighteq (v \mathbin{/_{l.f}} H)$	if $l \neq l' \vee f \neq f'$	TSSC4
$(u \trianglelefteq l.f.m \trianglerighteq v) \mathbin{/_{l.f}} H = l.\mathsf{tau} \circ (u \mathbin{/_{l.f}} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{T}$	TSSC5
$(u \trianglelefteq l.f.m \trianglerighteq v) \mathbin{/_{l.f}} H = l.\mathsf{tau} \circ (v \mathbin{/_{l.f}} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{F}$	TSSC6
$(u \trianglelefteq l.f.m \trianglerighteq v) \mathbin{/_{l.f}} H = \mathsf{D}$	if $H(\langle m \rangle) = \mathsf{B}$	TSSC7

Table 25
Axioms for thread-private-service composition

$\mathsf{S} \mathbin{/_{\mathsf{t}}} H = \mathsf{S}$		TPSC1
$\mathsf{D} \mathbin{/_{\mathsf{t}}} H = \mathsf{D}$		TPSC2
$(\mathsf{tau} \circ x) \mathbin{/_{\mathsf{t}}} H = \mathsf{tau} \circ (x \mathbin{/_{\mathsf{t}}} H)$		TPSC3
$(x \trianglelefteq f.m \trianglerighteq y) \mathbin{/_{\mathsf{t}}} H = (x \mathbin{/_{\mathsf{t}}} H) \trianglelefteq f.m \trianglerighteq (y \mathbin{/_{\mathsf{t}}} H)$	if $f \neq \mathsf{t}$	TPSC4
$(x \trianglelefteq \mathsf{t}.m \trianglerighteq y) \mathbin{/_{\mathsf{t}}} H = \mathsf{tau} \circ (x \mathbin{/_{\mathsf{t}}} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{T}$	TPSC5
$(x \trianglelefteq \mathsf{t}.m \trianglerighteq y) \mathbin{/_{\mathsf{t}}} H = \mathsf{tau} \circ (y \mathbin{/_{\mathsf{t}}} \frac{\partial}{\partial m} H)$	if $H(\langle m \rangle) = \mathsf{F}$	TPSC6
$(x \trianglelefteq \mathsf{t}.m \trianglerighteq y) \mathbin{/_{\mathsf{t}}} H = \mathsf{D}$	if $H(\langle m \rangle) = \mathsf{B}$	TPSC7
$(x \trianglelefteq \mathsf{mg}(n) \trianglerighteq y) \mathbin{/_{\mathsf{t}}} H = (x \mathbin{/_{\mathsf{t}}} H) \trianglelefteq \mathsf{mg}(n) \trianglerighteq (y \mathbin{/_{\mathsf{t}}} H)$		TPSC8

Table 25. In this table, f stands for an arbitrary focus from $\mathcal{F}$, m stands for an arbitrary method from $\mathcal{M}$, and n stands for an arbitrary natural number. Axiom TPSC3 shows that the action tau is always accepted. Axioms TPSC5 and TPSC6 make it clear that tau arises as the residue of processing unlocated actions. Therefore, tau is not connected to a particular focus and is always accepted. Axiom TPSC7 expresses that deadlock takes place when an action is not accepted. Axiom TPSC8 makes it clear that thread-private-service composition is not affected by thread migration.

Let T stand for either $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lck}}$, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbs}}$ or $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{cs}}$. Then we will write $T + \mathrm{TSC}$ for T extended with the thread-shared-service composition operators, the thread-private-service composition operator, and the axioms from Tables 24 and 25.

In $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}+\mathrm{TSC}$, we do not have something like $\|(\beta_1 \frown [\langle x \rangle_F^c]_l \frown \beta_2) \mathbin{/_{l.f}} H = \|(\beta_1 \frown [\langle x \mathbin{/_f} H \rangle_F^c]_l \frown \beta_2)$.[4] This fails because both explicit migration and load balancing may add threads to the local thread vector at location l. For similar reasons, we do not have such equations in $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}+\mathrm{TSC}$ or $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbs}}+\mathrm{TSC}$ either. This shows the key difference with the multi-level interleaving strategies from [8,10].

[4] The right-hand side of this equation is not a legitimate term from the language of $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}+\mathrm{TSC}$. The addition of the useless thread-shared-service operators $/_f$ is simple.

Table 26
Additional axioms for thread forking

$\|([\langle x \trianglelefteq \mathsf{nt}(z) \trianglerighteq y\rangle_F^c \frown \alpha]_l \frown \beta) = l.\mathsf{tau} \circ \|(pv([\langle x\rangle_F^{ic(c)} \frown \alpha \frown \langle z\rangle_\emptyset^0]_l \frown \beta))$	CDIlba10
$(x \trianglelefteq \mathsf{nt}(z) \trianglerighteq y) \mathbin{/_\mathsf{t}} H = (x \mathbin{/_\mathsf{t}} H) \trianglelefteq \mathsf{nt}(z \mathbin{/_\mathsf{t}} H) \trianglerighteq (y \mathbin{/_\mathsf{t}} H)$	TPSC9
$\pi_{n+1}(x \trianglelefteq \mathsf{nt}(z) \trianglerighteq y) = \pi_n(x) \trianglelefteq \mathsf{nt}(\pi_n(z)) \trianglerighteq \pi_n(y)$	P4

13 Thread Forking

The presence of thread forking adds to the need for load balancing, for the length of some local thread vectors may in that case increase solely by thread forking. In this section, we introduce a basic form of thread forking. We will do so like in [6].

We introduce the ternary *forking postconditional composition* operator $\_ \trianglelefteq \mathsf{nt}(\_) \trianglerighteq \_ : \mathbf{T} \times \mathbf{T} \times \mathbf{T} \to \mathbf{T}$. The forking postconditional composition operator has the same shape as the other postconditional composition operators. Formally, no action is involved in forking postconditional composition. However, for an operational intuition, in $p \trianglelefteq \mathsf{nt}(r) \trianglerighteq q$, $\mathsf{nt}(r)$ can be considered a thread forking action. It represents the act of forking off thread r. Like with real actions, a reply is produced. We consider the case where forking off a thread will never be blocked or fail. In that case, it always produces a positive reply. The action tau arises as the residue of forking off a thread.

If thread forking is added to $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$, then one additional axiom is needed: axiom CSIlba10 from Table 26. If thread forking is added to $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$+TSC, then two additional axioms are needed: axioms CSIlba10 and TPSC9 from Table 26. If thread forking is added to $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lba}}$+TSC+AIP, then three additional axioms are needed: axioms CSIlba10, TPSC9 and P4 from Table 26. If thread forking is added to $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lck}}$, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbb}}$, $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{lbs}}$ or $\mathrm{TA}_{\mathrm{dsi}}^{\mathrm{cs}}$, then an obvious variation on axiom CDIlba10 is needed.

In [6], we treat several interleaving strategies for threads that support a basic form of thread forking. All of them deal with cases where forking may be blocked and/or may fail. We believe that perfect forking is a suitable abstraction when dealing primarily with load balancing. In [6], $\mathsf{nt}(r)$ was formally considered a thread forking action. We experienced afterwards that this leads to unnecessary complications in expressing definitions and results concerning models for thread algebras featuring thread forking.

14 Conclusions

In [6], we have pursued the object to develop a theory about threads, interleaving of threads and interaction of threads with services that is useful for:

(i) gaining insight into the semantic issues concerning the multi-threading related features found in contemporary object-oriented programming languages such as Java and C#; (ii) simplified formal description and analysis of programs in which multi-threading is involved. In [9], we have extended the theory developed in [6] with features that allow for details of multi-threading that come up where it is intertwined with object-orientation to be dealt with. In this paper, we have extended the theory developed in [6] with features that allow for details that come up with distributed multi-threading to be dealt with. The features include explicit thread migration, load balancing and capability searching. We have taken load balancing and capability searching for implicit forms of thread migration that are part of distributed interleaving strategies.

To our knowledge, there is no other work on the theory of threads and multi-threading that covers distributed multi-threading. Moreover, we are not aware of other work on the theory of threads and multi-threading that is based on strategic interleaving. Although a deterministic interleaving strategy is always used for thread interleaving, it is the practice in work in which the semantics of multi-threaded programs is involved to look upon thread interleaving as arbitrary interleaving, see e.g. [1,13].

The work presented in this paper can be extended in several directions. We mention only malcode migration and implicit migration in grids. We think of an extension in the direction of malcode migration that opens a semantical perspective on issues of computer viruses. It is customary to look upon computer viruses from a syntactical perspective, see e.g. [11,12]. We agree with [22] that being a computer virus is basically a property of behaviours of programs and hence semantical by nature. We think of an extension in the direction of implicit migration in grids that results in a theory relevant to grid computing [15,16]. A great deal of work in the field of grid computing has been done on the development of software for initial implementations of a grid. Little work has been done on the development of mathematical models and theories relevant to grid computing. We believe that an extension of the work presented in this paper in the direction of implicit migration in grids might be a useful contribution to the field of grid computing. Such an extension might call for data interchange between threads distributed over different locations to be taken into account.

References

[1] E. Ábrahám, F. S. de Boer, W. P. de Roever, M. Steffen, A compositional operational semantics for JavaMT, in: N. Dershowitz (Ed.), Verification: Theory and Practice, Vol. 2772 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 290–303.

[2] J. C. M. Baeten, W. P. Weijland, Process Algebra, Vol. 18 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Cambridge, 1990.

[3] J. A. Bergstra, I. Bethke, Polarized process algebra and program equivalence, in: J. C. M. Baeten, J. K. Lenstra, J. Parrow, G. J. Woeginger (Eds.), Proceedings 30th ICALP, Vol. 2719 of Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 1–21.

[4] J. A. Bergstra, J. W. Klop, Process algebra: Specification and verification in bisimulation semantics, in: M. Hazewinkel, J. K. Lenstra, L. G. L. T. Meertens (Eds.), Proceedings Mathematics and Computer Science II, Vol. 4 of CWI Monograph, North-Holland, 1986, pp. 61–94.

[5] J. A. Bergstra, M. E. Loots, Program algebra for sequential code, Journal of Logic and Algebraic Programming 51 (2) (2002) 125–156.

[6] J. A. Bergstra, C. A. Middelburg, Thread algebra for strategic interleaving, to appear in *Formal Aspects of Computing*. Preliminary version: Computer Science Report 04-35, Department of Mathematics and Computer Science, Eindhoven University of Technology.

[7] J. A. Bergstra, C. A. Middelburg, Splitting bisimulations and retrospective conditions, Information and Computation 204 (7) (2006) 1083–1138.

[8] J. A. Bergstra, C. A. Middelburg, Thread algebra with multi-level strategies, Fundamenta Informaticae 71 (2/3) (2006) 153–182.

[9] J. A. Bergstra, C. A. Middelburg, A thread calculus with molecular dynamics, Computer Science Report 06-24, Department of Mathematics and Computer Science, Eindhoven University of Technology (August 2006).

[10] J. A. Bergstra, C. A. Middelburg, A thread algebra with multi-level strategic interleaving, Theory of Computing Systems 41 (1) (2007) 3–32.

[11] F. Cohen, Computer viruses – theory and experiments, Computers and Security 6 (1987) 22–35.

[12] F. Cohen, A Short Course on Computer Viruses, 2nd Edition, John Wiley and Sons, New York, 1994.

[13] C. Flanagan, S. N. Freund, S. Qadeer, S. A. Seshia, Modular verification of multithreaded programs, Theoretical Computer Science 338 (1/3) (2005) 153–183.

[14] W. J. Fokkink, Introduction to Process Algebra, Texts in Theoretical Computer Science, An EATCS Series, Springer-Verlag, Berlin, 2000.

[15] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, 1999.

[16] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: Enabling scalable virtual organizations, Journal of High Performance Computing Applications 15 (3) (2001) 200–222.

[17] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, 2nd Edition, Addison-Wesley, Reading, MA, 2000.

[18] A. Hejlsberg, S. Wiltamuth, P. Golde, C# Language Specification, Addison-Wesley, Reading, MA, 2003.

[19] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, 1985.

[20] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, 1989.

[21] D. Sannella, A. Tarlecki, Algebraic preliminaries, in: E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner (Eds.), Algebraic Foundations of Systems Specification, Springer-Verlag, Berlin, 1999, pp. 13–30.

[22] H. Thimbleby, S. Anderson, P. Cairns, A framework for modelling trojans and computer virus infection, Computer Journal 41 (7) (1999) 444–458.

[23] M. Wirsing, Algebraic specification, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B, Elsevier, Amsterdam, 1990, pp. 675–788.