INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

An open structural operational semantics for an object-oriented calculus with thread classes

Erika Ábrahám Andreas Grüner
Martin Steffen

Bericht Nr. 0505
May 13, 2005

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

An open structural operational semantics for an object-oriented calculus with thread classes

Erika Ábrahám Andreas Grüner
Martin Steffen

Bericht Nr. 0505
May 13, 2005

e-mail: eab@informatik.uni-freiburg.de,
{ang|ms}@informatik.uni-kiel.de

Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1/2/4).

An open structural operational semantics for an object-oriented calculus with thread classes*

May 13, 2005

Erika Ábrahám[2] and Andreas Grüner[1] and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] Albert-Ludwigs-University Freiburg, Germany

Abstract. In this report we present a multithreaded class-calculus featuring *thread classes*.
From an observational point of view, considering classes as part of a component makes instantiation a possible interaction between component and environment or observer. For thread classes it means that a component may create external activity, which influences what can be observed. The fact that cross-border instantiation is possible requires that the *connectivity* of the objects needs to be incorporated into the semantics. We extend our prior work not only by adding thread classes, but also in that thread names may be *communicated,* which means that the semantics needs explicitly account for the possible acquaintance of objects with threads.
This report formalizes a calculus featuring thread classes, i.e., its syntax, type system, and operational semantics. We furthermore discuss observational aspects of thread classes.

Keywords: class-based object-oriented languages, open systems, formal semantics, thread classes, heap abstraction.

* Part of this work has been financially supported by the IST project Omega (IST-2001-33522) and the NWO/DFG project Mobi-J (RO 1122/9-1/2/4).

Table of Contents

1 Introduction

The foremost role of classes in an object-oriented language is that they are *"generators of state".* In an observational framework, and distinguishing between the component under observation and an observing environment, this makes object instantiation a possible component-environment interaction. As a consequence, a faithful representation of the observational behavior of class-structured components requires to represent the connectivity among objects in the semantics, which can be seen as a worst-case approximation of the heap's reference structure [3] [4].

In languages like *Java* [9] and $C^{\#}$ [7], objects are passive entities; the active part of the program is represented by threads. Indeed, in a multi-threaded setting, there is also a mechanism for *"generating new activity"*, i.e., for creating new threads. In this paper we extend our previous work by thread instantiation from *classes.* In [4], we concentrated on a single-threaded fragment, while [3] was multi-threaded, but without thread classes, i.e., new activities could be dynamically spawned but not from "templates". This extension makes cross-border activity generation a possible component-environment interaction, i.e., the component may create threads in the environment and vice versa. Without thread classes (but ordinary classes), only cross-border generation of objects was possible.

This generalization makes the semantics account more resembling the situation as for instance in *Java*, it complicates the semantics, however, since now also the connectivity of threads has to be taken into account.

Overview The paper is organized as follows. Section 2 contains syntax and operational semantics of the calculus we use, formalizing the notion of thread classes. Section 3 concludes with related and future work.

2 A multithreaded calculus with thread classes

Next we present the calculus, and we start with the syntax. It is based on the multithreaded object calculus, similar to the one presented in [8] and in particular [10]. Compared to our previous work for instance in [2], we added thread classes as generators of activity.

2.1 Syntax

The abstract syntax is given in Table 1. A program is given by a collection of classes where a class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields. *Thread classes,* written $c_t\langle\!\langle t_a \rangle\!\rangle$, is known under the name c_t and carries the code in t_a. For names, we will generally use o and its syntactic variants as names for objects, c for classes (in particular c_t for thread classes), and n when being unspecific, for instance in Table 1.

An object $o[c, F]$ stores the current value of the fields or instance variables and keeps a reference to the class it instantiates. A method $\varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. Besides named objects and classes, the dynamic configuration of a program contains threads $n\langle t \rangle$ as active entities.

A thread basically is either a value or a sequence of expressions, notably method calls (written $v.l(\vec{v})$), the creation of new objects *new* c where c is a class name, and *thread instantiation* written as *spawn* $c_t(\vec{v})$.

Furthermore we will use f for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update[3] as $x.f := v$.

The available types are given in the following grammar:

$$
\begin{aligned}
T &::= B \mid \mathit{thread} \mid n \\
U &::= T \times \ldots \times T \to T \\
V &::= T \mid U \mid [l{:}U, \ldots, l{:}U] \mid [\![l{:}U, \ldots, l{:}U]\!] \mid \mathit{none}
\end{aligned}
$$

Besides base types B if wished, the type *thread* denotes the type of thread names, and *none* represents the absence of a return value. The name n of a class serves as the type for the named instances of the class. Finally we need for the type system, i.e., as auxiliary type construction, the type or interface of unnamed objects, written $[l_1{:}U_1, \ldots, l_k{:}U_k]$ and the type for classes, written $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$.

[3] We don't use general method update as in the object-based calculus.

C	$::= \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F] \mid n\langle t\rangle \mid n\langle\!\langle t_a \rangle\!\rangle$	program
O	$::= F, M$	object
M	$::= l = m, \ldots, l = m$	method suite
F	$::= l = f, \ldots, l = f$	fields
m	$::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
f	$::= \varsigma(n{:}T).\lambda().v$	field
t_a	$::= \lambda(x{:}T, \ldots, x{:}T).t$	thread abstraction
t	$::= v \mid \mathit{stop} \mid \mathit{let}\, x{:}T = e\, \mathit{in}\, t$	thread
e	$::= t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e$	expr.
	$\mid\ v.l(v, \ldots, v) \mid v.l := v \mid \mathit{currentthread}$	
	$\mid\ \mathit{new}\, n \mid \mathit{spawn}\, n(v, \ldots, v)$	
v	$::= x \mid n$	values

Table 1. Abstract syntax

2.2 Type system

The type system or static semantics presented next characterizes the well-typed programs. The derivation rules are shown in Tables 2 and 3.

Table 2 defines the typing on the level of global configurations, i.e., on "sets" of objects and classes, all named, together with the threads. On this level, the typing judgments are of the form

$$\Delta \vdash C : \Theta \ ,$$

where Δ and Θ are finite mappings from names to types. In the judgment, Δ plays the role of the typing assumptions about the environment, and Θ the commitments of the configuration, i.e., the names offered to the environment. Sometimes, the words required and provided interface are used to describe the dual roles. Δ must contain at least all external names referenced by C and dually Θ mentions at most the names offered by C. For a pair Δ and Θ of assumption and commitment context to be *well-formed* we furthermore require that the domains of Δ and Θ are disjoint except for thread names.

The empty configuration is denoted by $\mathbf{0}$; it is well-typed in any context and exports no names (cf. rule T-Empty). Two configurations in parallel can refer mutually to each other's commitments, and together offer the union of their names (cf. rule T-Par). It will be an invariant of the operational semantics that the identities of parallel entities are disjoint. Therefore, Θ_1 and Θ_2 in the rule for parallel composition are merged disjointly, which is indicated by writing $\Theta_1 + \Theta_2$. For the assumption contexts, Δ, Θ_1 respectively Δ, Θ_2 is meant to denote disjoint union except thread names.

Remark 1 (Thread names and parallel composition). Note that T-Par does not allow a thread name to occur on both sides of the parallel composition. The typing excludes terms of the form $n\langle t_1\rangle \parallel n\langle t_2\rangle$ as part of the component. Indeed,

the operational semantics will not need to consider the behavior of the parallel composition of a thread with itself. □

The ν-binder hides the bound name inside the component (cf. the rules T-NU$_i$ and T-NU$_e$). All names can be hidden, i.e., class names, in particular names of thread classes, as well as object and thread references. For class names, which are never transmitted, the ν-binder acts statically, i.e., a class name under a binder remains permanently hidden.

The two variants of the rule distinguish basically the situation of hiding for lazy instantiation from all other forms of hiding. Since the instance of a class always belongs to the part of the system, where its class resides, the new name is added in case of lazy instantiation (cf. rule T-NU$_e$) to the environment context; otherwise the new name is added to the commitment context. Note that there is no special treatment of cross-border thread instantiation, for instance in a rule similar to T-NU$_e$. The reason is that threads are not instantiated lazily. To put it differently: there are no terms of the form $\nu(n{:}c_t).C$ where c_t is a thread class of the environment. When instantiating a thread class of the environment, the scope is immediately opened. Possible are only components of the form $\nu(n{:}\mathit{thread}).C$, which results from internal thread creation.

For both T-NU-rules, the ν-construct does not only introduce a local scope for its bound name but asserts something stronger, namely the *existence* of a likewise named entity. This highlights one difference of let-bindings for variables and the introduction of names via the ν-operator: the language construct to introduce names is the *new*-operator, which opens a new local scope and a named component "running in parallel". We call the fact that object references of external objects can be introduced and instantiated only later when first used, *lazy instantiation;* see Section 2.3 for the operational behavior.

Let-bound variables are *stack* allocated and checked in a stack-organized variable context Γ. Names created by *new* are *heap* allocated and thus checked in a "parallel" context (cf. again the assumption-commitment rule T-PAR). The rules for named classes introduce the name of the class and its type into the commitment (cf. T-NCLASS and T-NTCLASS); The code of the class $[\![O]\!]$ respectively the code of the thread class $\langle\!\langle t_a \rangle\!\rangle$ is checked in an assumption context where the name of the class is available.

An instantiated object will be available in the exported context Θ by rule T-NOBJ. Running threads are treated similarly, except that they possess as type not the name of their thread class, but the type *none*, which expresses that they do not return with a value.[4]

Remark 2 (Thread classes and types). Thread classes and ordinary classes are treated slightly differently as far as the typing is concerned. Where for objects, the name of its class is taken as the type, threads have the general *thread* as their type. The reason for that decision is partly technical. It would be straightforward,

[4] For the thread in T-NTHREAD, the type *none* can be generated by the atomic thread *stop*. In principle, a variable could have the type *none*, as well, but there are no values except variables of this type.

to use also for threads their class name as type. From a pragmatic point of view, this seems unplausibly restrictive, however. On the other hand, we decided for the simple scheme for typing objects, to leave aside the orthogonal issue of subtyping and inheritance. For the types of threads, subtyping is not an issue, since one cannot "do" anything with a thread name (for instance communicating with the thread) except comparing it with other names. In particular, a thread cannot observe the fact that it is an instance of a particular class. *Java*, for instance, connects the notions of objects and threads in such a way, that the instance of a "thread class" is an object in which the thread starts its life, the thread could determine the identity of its thread class via the `instanceof`-operator. □

The last rule is a rule of *subsumption*. It expresses a very simple form of subtyping: we allow that an object respectively class contains *at least* the members than the interface requires. This corresponds to width subtyping. Note, however, that each object has exactly one type, namely its class.

Definition 1 (Subtyping). *Let Δ_1 and Δ_2 be two well-formed name contexts. Then $\Delta_1 \leq \Delta_2$, if Δ_1 and Δ_2 have the same domain, and additionally $\Delta_1(n) \leq \Delta_2(n)$ for all names. In abuse of notation, the relation $\leq$ on types is defined as identity for all types except for object interfaces where we have:*

$$[\![l_1{:}T_1, \ldots, l_k{:}T_k, l_{k+1}{:}T_{k+1}, \ldots]\!] \leq [\![l_1{:}T_1, \ldots l_k{:}T_k]\!] \ .$$

The relations $\leq$ are obviously reflexive, transitive, and antisymmetric.

The typing rules of Table 3 formalize typing judgments for threads and objects and their syntactic sub-constituents. Besides assumptions about the provided names of the environment kept in Δ as before, the typing is done relative to assumptions about occurring free variables. They are kept separately in a variable context Γ, a finite mapping from variables to types.

The typing rules are rather straightforward and in many cases identical to the ones from [10] and [2]. We allow ourselves to write $\vec{T}$ and $\vec{v}$ for $T_1 \times \ldots \times T_k$ and $v_1, \ldots, v_k$ and similar abbreviations, where we assume that the number of arguments match in the rules. Different from the object-based setting are the ones dealing with objects and classes. Rule T-Class is the introduction rule for class types, the rule of instantiation of a class T-NewC requires reference to a class-typed name. Similarly for thread classes, which are typed as functions from the domain of their constructor to the domain of threads in rule T-TClass. Consequently, the spawning of a new thread yields an element of *thread*, if the type of the actual parameters match with the required ones. Note also that the deadlocking expression *stop* has every type.

2.3 Operational semantics

Next we present the operational semantics; it is given in two stages. Section 2.3.1 starts with component-internal steps, i.e., those definable without reference to the environment. In particular, the steps have no observable external effect and are formulated independently of the assumption and commitment contexts.

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-EMPTY} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \qquad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1 + \Theta_2}\ \text{T-PAR}$$

$$\frac{\Delta \vdash C : \Theta, n{:}T \qquad \Delta \not\vdash T : [\![\ldots]\!]}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \text{T-NU}_i \qquad \frac{\Delta, o{:}c \vdash C : \Theta \qquad \Delta \vdash c : [\![\ldots]\!]}{\Delta \vdash \nu(o{:}c).C : \Theta}\ \text{T-NU}_e$$

$$\frac{;\, \Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NCLASS} \qquad \frac{;\, \Delta, c_t{:}T \vdash \langle\!\langle t_a \rangle\!\rangle : T}{\Delta \vdash c_t\langle\!\langle t_a \rangle\!\rangle : (c_t{:}T)}\ \text{T-NTCLASS}$$

$$\frac{;\, \Delta \vdash c : [\![T_F, T_M]\!] \qquad ;\, \Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F] : (o{:}c)}\ \text{T-NOBJ}$$

$$\frac{;\, \Delta, n{:}\,\mathit{thread} \vdash t : \mathit{none}}{\Delta \vdash n\langle t \rangle : (n{:}\,\mathit{thread})}\ \text{T-NTHREAD}$$

$$\frac{\Delta' \leq \Delta \qquad \Theta \leq \Theta' \qquad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-SUB}$$

Table 2. Static semantics (components)

The external steps, on the other hand, presented in Section 2.3.2, define the interaction of the component with the environment. In particular, the semantics is defined in reference to assumption and commitment contexts. The static part of the contexts corresponds to the type system from Section 2.2 on component level and takes care that, e.g., only well-typed values are received from the environment. The contexts, however, need to be extended by a *dynamic* part which deals with the potential *connectivity* of objects and thread names and which corresponds to an abstraction of the heap of the program.

2.3.1 Internal steps The internal steps are given in Table 4, where we distinguish between confluent steps, written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$. The first 5 rules deal with the basic sequential constructs, all as $\rightsquigarrow$-steps. The basic evaluation mechanism is substitution (cf. rule RED). Note that the rule requires that the leading let-bound variable of a thread can be replaced only by *values*. This means the redex (if any) is uniquely determined within the thread which makes the reduction strategy deterministic. The *stop*-thread terminates for good, i.e., the rest of the thread will never be executed (cf. rule STOP).

The step NEWO_i describes the creation of an instance of a component *internal* class $c[\![F, M]\!]$, i.e., a class whose name is contained in the configuration. Note that instantiation is a confluent step. The fields F of the class are taken as template for the created object, and the identity of the object is new and

$$\frac{\Gamma;\Delta \vdash m_1 : T_1 \quad \ldots \quad \Gamma;\Delta \vdash m_k : T_k \qquad T = [\![l_1{:}T_1, \ldots, l_k{:}T_k]\!]}{\Gamma;\Delta \vdash [\![l_1 = m_1, \ldots, l_k = m_k]\!] : T}\ \text{T-Class}$$

$$\frac{\Gamma;\Delta \vdash f_1 : T_1 \quad \ldots \quad \Gamma;\Delta \vdash f_k : T_k \qquad T = [l_1{:}T_1, \ldots, l_k{:}T_k]}{\Gamma;\Delta \vdash [l_1 = f_1, \ldots, l_k = f_k] : T}\ \text{T-Obj}$$

$$\frac{\Gamma, x_1{:}T_1, \ldots, x_k{:}T_k; \Delta \vdash t : none}{\Gamma;\Delta \vdash \langle\!\langle \lambda(\vec{x}{:}\vec{T}).t \rangle\!\rangle : \vec{T} \to thread}\ \text{T-TClass}$$

$$\frac{\Gamma, x_1{:}T_1, \ldots, x_k{:}T_k; \Delta, n{:}c \vdash t : T' \quad \Gamma;\Delta \vdash c : T \quad T = [\![\ldots, l{:}\vec{T} \to T', \ldots]\!]}{\Gamma;\Delta \vdash \varsigma(n{:}c).\lambda(\vec{x}{:}\vec{T}).t : T.l}\ \text{T-Memb}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad \Gamma;\Delta \vdash \vec{v} : \vec{T}}{\Gamma;\Delta \vdash v.l(\vec{v}) : T}\ \text{T-Call}$$

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : T \qquad \Gamma;\Delta \vdash v' : T.f}{\Gamma;\Delta \vdash v.f := v' : c}\ \text{T-FUpdate}$$

$$\frac{\Gamma;\Delta \vdash c : [\![T]\!]}{\Gamma;\Delta \vdash new\ c : c}\ \text{T-NewC} \qquad \frac{\Gamma;\Delta \vdash n : \vec{T} \to thread \quad \Gamma;\Delta \vdash \vec{v} : \vec{T}}{\Gamma;\Delta \vdash spawn\ n(\vec{v}) : thread}\ \text{T-Spawn}$$

$$\frac{}{\Gamma;\Delta \vdash currentthread : thread}\ \text{T-CurrT}$$

$$\frac{\Gamma;\Delta \vdash e : T_1 \qquad \Gamma, x{:}T_1; \Delta \vdash t : T_2}{\Gamma;\Delta \vdash let\ x{:}T_1 = e\ in\ t : T_2}\ \text{T-Let}$$

$$\frac{\Gamma;\Delta \vdash v_1 : T_1 \qquad \Gamma;\Delta \vdash v_2 : T_1 \qquad \Gamma;\Delta \vdash e_1 : T_2 \qquad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-Cond}$$

$$\frac{}{\Gamma;\Delta \vdash stop : T}\ \text{T-Stop} \qquad \frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-Var} \qquad \frac{\Delta(n) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-Name}$$

Table 3. Static semantics (2)

$n\langle let\, x{:}T = v\, in\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle$ RED

$n\langle let\, x_2{:}T_2 = (let\, x_1{:}T_1 = e_1\, in\, e)\, in\, t\rangle \rightsquigarrow$
$\quad n\langle let\, x_1{:}T_1 = e_1\, in\, (let\, x_2{:}T_2 = e\, in\, t)\rangle$ LET

$n\langle let\, x{:}T = (\mathsf{if}\, v = v\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, in\, t\rangle \rightsquigarrow n\langle let\, x{:}T = e_1\, in\, t\rangle$ $\textsc{Cond}_1$

$n\langle let\, x{:}T = (\mathsf{if}\, v_1 = v_2\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, in\, t\rangle \rightsquigarrow n\langle let\, x{:}T = e_2\, in\, t\rangle$ $\textsc{Cond}_2$

$n\langle let\, x{:}T = stop\, in\, t\rangle \rightsquigarrow n\langle stop\rangle$ STOP

$n\langle let\, x{:}T = currentthread\, in\, t\rangle \rightsquigarrow n\langle let\, x{:}T = n\, in\, t\rangle$ CURRENTTHREAD

$c[\![F, M]\!] \parallel n\langle let\, x{:}T = new\, c\, in\, t\rangle \rightsquigarrow$
$\quad c[\![F, M]\!] \parallel \nu(o{:}T).(o[c, F] \parallel n\langle let\, x{:}c = o\, in\, t\rangle)$ $\textsc{NewO}_i$

$c_t\langle\!\langle \lambda(\vec{x}{:}\vec{T}).t_2\rangle\!\rangle \parallel n_1\langle let\, x{:}T = spawn\, c_t(\vec{v})\, in\, t_1\rangle \rightsquigarrow$
$\quad c_t\langle\!\langle \lambda(\vec{x}{:}\vec{T}).t_2\rangle\!\rangle \parallel \nu(n_2{:}T).(n_1\langle let\, x{:}T = n_2\, in\, t_1\rangle \parallel n_2\langle t_2[\vec{v}/\vec{x}]\rangle)$ $\textsc{Spawn}_i$

$c[\![F, M]\!] \parallel o[c, F'] \parallel n\langle let\, x{:}T = o.l(\vec{v})\, in\, t\rangle \xrightarrow{\tau}$
$\quad c[\![F, M]\!] \parallel o[c, F'] \parallel n\langle let\, x{:}T = M.l(o)(\vec{v})\, in\, t\rangle$ $\textsc{Call}_i$

$o[c, F] \parallel n\langle let\, x{:}T = o.f := v\, in\, t\rangle \xrightarrow{\tau} o[c, F.f := v] \parallel n\langle let\, x{:}T = o\, in\, t\rangle$ FUPDATE

Table 4. Internal steps

local —for the time being— to the instantiating thread; the new named object and the thread are thus enclosed in a ν-binding. Similarly, rule $\textsc{Spawn}_i$ specifies internal thread class instantiation.

Rule $\textsc{Call}_i$ treats an internal method call, i.e., a call to an object contained in the configuration. In the step, $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when method suite $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Note also that the step is a $\xrightarrow{\tau}$-step, not a confluent one. The same holds for field update in rule FUPDATE, where $[c, (l_1 = f_1, \ldots, l_k = f_k, f = v').f := v]$ stands for $[c, l_1 = f_1, \ldots, l_k = f_k, f = v]$. Note further that instances of a component class invariantly belong to the component and not to the environment. This means that an instance of a component class resides after instantiation in the component, and named objects will never be exported from the component to the environment or vice versa; of course, *names* to objects may well be exported.

The reduction relations from above are used modulo *structural congruence,* which captures the algebraic properties of parallel composition and the hiding operator. The basic axioms for $\equiv$ are shown in Table 5 where in the fourth axiom, n does not occur free in C_1. The congruence relation is imported into the reduction relations in Table 6. Note that all syntactic entities are always tacitly understood modulo α-conversion.

$$\mathbf{0} \parallel C \equiv C$$
$$C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2)$$
$$\nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Table 5. Structural congruence

2.3.2 External steps Besides internal steps a component exchanges information with the environment via *calls, returns,* and *spawn* actions (cf. Table 7). In the call and return labels, the mentioned n is the active thread that issues the call or returns from the call. In the thread instantiation label, n is the name of the new thread; the thread which spawned the new thread is *not* part of the label.[5] In accordance with π-calculus terminology, let us say, the thread name occurs in the label in *subject* position in the first case and in *object* or argument position in the latter. Of course, a thread name may occur in both positions at the same time. Furthermore note that there are no separate external labels for object instantiation: Externally instantiated objects are created only at the point when it is actually accessed for the first time, which we call "*lazy instantiation*". Given a label $\nu(\Phi).\gamma'$ where Φ is a name context, i.e., a sequence of single $\nu(n{:}T)$ bindings (whose names are assumed all disjoint, as usual) and were γ' does not contain any binders, we call γ' the *core* of the label. Given a label γ, we refer with $\lfloor\gamma\rfloor$ to its core. Analogously for send and receive labels.

Augmentation To formulate the external communication properly, we need to introduce a few augmentations. We extend the syntax by two additional expressions

$$o_1 \ \mathit{blocks\ for}\ o_2 \quad \text{and} \quad o_2 \ \mathit{return\ to}\ o_1\ v\ .$$

[5] Of course it might be mentioned in the arguments.

$$\frac{C \equiv \rightsquigarrow \equiv C'}{C \rightsquigarrow C'} \qquad \frac{C \rightsquigarrow C'}{C \parallel C'' \rightsquigarrow C' \parallel C''} \qquad \frac{C \rightsquigarrow C'}{\nu(n{:}T).C \rightsquigarrow \nu(n{:}T).C'}$$
$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n{:}T).C \xrightarrow{\tau} \nu(n{:}T).C'}$$

Table 6. Reduction modulo congruence

$\gamma ::= n\langle call\ o.l(\vec{v})\rangle \mid n\langle return(v)\rangle \mid \langle spawn\ n\ of\ c(\vec{v})\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	receive and send

Table 7. Labels

The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1. Furthermore, we augment the syntax of the method definitions accordingly, such that each method call and each spawn step is preceded by an annotation of the caller; i.e., instead of $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots)$ we write

$$\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots self\, x.l(\vec{y})\ldots self\ spawn\, c_t(\vec{z}))\ .$$

Even if threads themselves cannot communicate, their names can be communicated via message passing. To obtain a faithful representation of the behavior, the semantics must contain information to which clique of objects the thread belongs to. Especially for *new* threads, the semantics needs a representation of that clique. Note that a thread can be instantiated *without* connection to any object/clique and indeed the initial thread starts with static code, i.e., without reference to any object. As representation of the clique of objects, the thread n starts in $\odot_n$. As said, $\odot_n$ may not correspond to any existing object; we need this representative just to maintain the connectivity of the thread in case there is indeed no (visible) object.

We need to augment the threads such that every thread n carries at the beginning the identity $\odot_n$ of its initial clique. The program starts with one single initial thread. If the thread starts within the component, the contexts of the initial configuration $\Delta_0 \vdash C : \Theta_0$ asserts $\Theta_0 \vdash \odot$. Otherwise, $\Delta_0 \vdash \odot$. As in the augmentation for methods, the code in the thread classes must be augmented in such a way, that for method calls the virtual clique of the thread is mentioned in front of the call, i.e., after instantiation, the call looks as follows: $n\langle\ldots \odot_n\, x.l(\vec{v})\ldots\rangle$. The static code of each thread class is augmented into

$$c_t\langle\!\langle\lambda(\vec{x}{:}\vec{T}).(\ldots \odot\, x.l(\vec{v})\ldots)\rangle\!\rangle$$

for each mentioned call. When the thread is instantiated, $\odot$ is replaced by $\odot_n$ where n is the identity of the new thread. Given the above thread class, we denote by $c_t(\vec{v})$ the replacement $t[\odot_n, \vec{v}/\odot, \vec{x}]$, when t is the body of the thread class definition. The initial thread, which is not instantiated from a thread class but given directly (in case the activity starts in the component) starts with $\odot_n$ as augmentation, if the initial thread is named n. If the component is renamed by α-conversion, n and $\odot_n$ are renamed simultaneously. The steps of the internal semantics must be adapted accordingly. One particular thing we require for the treatment of *stop* in connection with the block-return augmentation: The internal rule Stop for the deadlocked thread is adapted insofar that it does *not* remove a o_s *return to* o_r v-statement. We also omit the typing rules for the augmentation, as they are straightforward.

Remark 3 (Augmentation). Intuitively, the introduction of the auxiliary symbols $\odot_n$ does not influence the behavior, since the programmer is not allowed to use them in his code. Note in this context that $\odot_n$ is *not* comparable with n. Note further that $\odot_n$ which we introduced for an accurate representation of the semantics is straightforwardly implementable in the given language. □

Remark 4 (Thread classes). In *Java*, thread classes are ordinary classes, i.e., classes which are instantiated into objects, which possess a special method that can be used to spawn a new thread. $\odot_n$ can be seen as analogue of that "thread object". A difference is, however, that in our setting, $\odot_n$ is not a real object, e.g., it is not included in the type system. In particular it does not contains fields nor methods, which means one cannot use $\odot_n$ to communicate information to the thread n. In our formalization, the only way the spawner of a new thread can hand over information to the spawnee is *explicitely* via the thread constructor. □

Connectivity contexts In the presence of cross-border instantiation, the semantics must contain a representation of the connectivity, which can be seen as an abstraction of the program's heap. The external semantics is formalized as labeled transitions between judgments of the form

$$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \ , \tag{1}$$

where $\Delta, \Sigma; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta, \Sigma; E_\Theta$ the *commitments*; alternative names are the required and the provided interface of the component. The assumptions consist of a part Δ, Σ concerning the existence (plus static typing information) of *named entities* in the environment. The semantics maintains as invariant that the assumption and commitment contexts are disjoint concerning object and class names, whereas a thread name occurs as assumption iff. it is mentioned in the commitments. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names.

This means, as invariant we maintain for all judgments $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ that Δ, Σ, and Θ are pairwise disjoint. A further invariant is that a thread name n occurs in Σ, iff. $\odot_n$ occurs in either Δ or else in Θ. This means, besides being relevant for connectivity information, $\odot_n$ contains also the information whether the threads started its life in the environment or in the component.

As mentioned, the $\odot_n$ are needed in particular because new thread names may be communicated between environment and component. If the thread has been active at the interface in the past, the semantics contains enough information such that the originating clique of objects is clear.

For the book-keeping of which objects of the environment have been told which identities, a well-typed component must take into account the *relation* of object from the assumption context Δ amongst each other, and the knowledge of objects from Δ about thread names and names exported by the component,

i.e., those from Θ. In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component:

$$E_\Delta \subseteq \Delta \times (\Delta + \Sigma + \Theta) \ . \tag{2}$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Sigma + \Delta)$. Since in the language we allow the sending of thread names, we must include pairs from $\Delta \times \Sigma$ resp. $\Theta \times \Sigma$ into the connectivity. We write $o \hookrightarrow n$ ("o may know n") for pairs from these relations. Without full information about the complete system, the component must make worst-case assumptions concerning the proliferation of knowledge, which are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of *objects from* Δ. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

$$\leftrightharpoons \triangleq \ (\hookrightarrow\!\downarrow_\Delta \cup \hookleftarrow\!\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ . \tag{3}$$

Note that we close $\hookrightarrow$ only wrt. environment objects, but not wrt. objects at the *interface* nor wrt. *thread* names, i.e., the part of $\hookrightarrow \subseteq \ \Delta \times (\Theta + \Sigma)$. The intuitive reason is that the closure expresses the worst-case assumptions about the environment behavior. The objects from Θ', however, are not under control of the environment. That the closure does not concern thread names reflects the fact that threads "themselves" cannot distribute information except by method calls, i.e., via objects. Threads do not communicate and exchange information, it's rather the objects that exchange information via method calls, which constitute the threads. We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Sigma + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\!\hookrightarrow$ for that union. As judgment, we use $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta, \Sigma$, resp. $\Delta, \Sigma; E_\Delta \vdash o \leftrightharpoons\!\!\hookrightarrow n : \Theta, \Sigma$. For Θ, Σ, E_Θ, and Δ, Σ, the definitions are applied dually.

The relation $\leftrightharpoons$ partitions the objects from Δ into equivalence classes. We call a set of object names from Δ (or dually from Θ) such that for all objects o_1 and o_2 from that set, $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta, \Sigma$, a *clique*, and if we speak of *the* clique of an object we mean the equivalence class.

Having introduced E_Δ and E_Θ as part of the judgment, we must still clarify what it "means", i.e., when does $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ hold? Besides the typing part, which remains unchanged, this concerns the commitment part E_Θ. The relation E_Θ asserts about the component C that the connectivity of the objects from the component is *not larger than* the connectivity entailed by E_Θ. Given a component C and two names o from Θ and n from $\Theta + \Delta + \Sigma$, we write $C \vdash o \hookrightarrow n$, if $C \equiv \nu(\Phi).(C' \parallel o[\ldots, f = n, \ldots])$ where o and n are not bound by Φ, i.e., o contains in one of its fields a reference to n. We can thus define:

Definition 2. *The judgment* $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ *holds, if* $\Delta, \Sigma \vdash C : \Theta, \Sigma$*, and if* $C \vdash n_1 \hookrightarrow n_2$*, then* $\Theta, \Sigma; E_\Theta \vdash n_1 \leftrightharpoons\!\!\hookrightarrow n_2 : \Delta, \Sigma$*.*

We often simply write $\Delta, \Sigma \vdash C : \Theta, \Sigma; E_\Theta$ to assert that the judgment is satisfied. Note that references mentioned in threads do not "count" as acquaintance.

The pairs listed in a commitment context E_Θ do not require the *existence* of connections in the components, it is rather the contrapositive situation: If E_Θ

does *not* imply that two entities are in connection, either directly or indirectly, then they must not be in connection in C. Thus, a larger E_Θ means a weaker specification. To make this precise, let us define what it means for one context to be stronger than another:

Definition 3 (Entailment). *$\Delta_1, \Sigma_1; E_{\Delta_1}; \Theta_1 \vdash \Delta_2, \Sigma_2; E_{\Delta_2}; \Theta_2$ iff. for all names n and n' with $\Delta_2 \vdash n$ and $\Delta_2 + \Sigma_2 + \Theta_2 \vdash n'$ we have: if $\Delta_2, \Sigma_2; E_{\Delta_2} \vdash n \leftrightharpoons\hookrightarrow n' : \Theta_2$, then $\Delta_1, \Sigma_1; E_{\Delta_1} \vdash n \leftrightharpoons\hookrightarrow n' : \Theta_1$.*

Note that since $\leftrightharpoons$ is reflexive on Δ_2, the above definition implies $\Delta_1 \geq \Delta_2$, by which we mean that the binding context Δ_1 is an extension of Δ_2 wrt. object names (analogously we write $\Delta_2 \leq \Delta_1$ when Δ_2 is extended by Δ_1, and say that Δ_2 is a *subcontext* of Δ_1).

As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ over-approximates the actual connectivity of the component and is update in incoming communication, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. Incoming new names, exchanged boundedly, however, update both commitments and assumptions.

Use and change of contexts The operational semantics is formulated as transitions between typed judgments

$$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \ .$$

The assumption contexts $\Delta, \Sigma; E_\Delta$ can be seen as an abstraction of the (not-present) environment; more precisely, it represents the potential behavior of all possible environments. So the assumption contexts are consulted to *check* whether an *incoming* action is currently possible, and are *updated* in outgoing communication. The commitment contexts play a dual role, i.e., it is updated in incoming communication. For outgoing communication, however, the commitment context is not consulted for checks.

Notation 1 *To facilitate the following definitions notationally, we will make use of the following conventions. We abbreviate the triple of name contexts Δ, Σ, Θ as Φ, and the context $\Delta, \Sigma, \Theta, E_\Delta, E_\Theta$ combining assumptions and commitments Ξ. Furthermore we understand $\acute{\Theta}, \acute{\Sigma}, \acute{\Delta}$ as $\acute{\Phi}$, $\acute{\Xi}$ as consisting of $\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta, \acute{E}_Delta, \acute{E}_Theta$ etc.*

The check whether the current assumptions are met in an incoming communication step is given in Definition 4.

Definition 4 (Connectivity check). *An incoming core label a with sender o_s and receiver o_r is* well-connected *wrt. an assumption-commitment context $\acute{\Xi}$ (written $\acute{\Xi} \vdash o_s \xrightarrow{a} o_r : ok$) if:*

$$\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash o_s \leftrightharpoons\hookrightarrow fn(a) : \acute{\Theta} \ . \tag{4}$$

Note that in case of an incoming *call* label, $fn(a)$ includes the receiver o_r and the thread name.

Besides *checking* whether the connectivity assumptions are met before a transition, the contexts are *updated* by a step, reflecting the change of knowledge. In first approximation, an incoming communication updates the commitment contexts, but not the assumption context, and dually for outgoing communication.

More precisely, however, incoming communication, for instance, updates *both* contexts, namely in connection with references exchanged boundedly. All external transitions may exchange *bound* names in the label, i.e., bound references to objects and threads, but not to classes since class names cannot be communicated. For the binding part $\Phi' = \Delta', \Sigma', \Theta'$ of a label $\nu(\Phi').\gamma$, we distinguish references to existing objects whose scope extrudes across the border, object names which are lazily instantiated in the step, and references to existing threads whose scope extrudes. In the special case of a spawn-label for cross-border instantiation of a new thread, also the new thread is transmitted boundedly, of course. Remember that for thread instantiation we cannot have lazy instantiation.

For incoming communication, with the binding part $\Phi' = \Delta', \Sigma', \Theta'$, the bindings Δ' and Σ' are object references respectively thread names transmitted by *scope extrusion,* and Θ' the reference to the *lazily instantiated* objects. For object references, the distinction is based on the class types which are never transmitted. In the incoming step, Δ' extends the assumption context Δ, Θ' extends Θ, and Σ' extends the assumption and the commitment context. For outgoing communication, the situation is dual. Cf. Definition 5.

Definition 5 (Name context update: Φ+a). *The* update $\acute{\Phi}$ *of an assumption-commitment context Φ wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor$ is defined as follows.*

1. $\acute{\Theta} = \Theta + \Theta'$. *In case of a spawn-label* $\acute{\Theta} = \Theta + \Theta', \odot_n$, *where n is the name of the spawned thread.*
2. $\acute{\Delta} = \Delta + \odot_{\Sigma'}, \Delta'$. *In case of a spawn label, $\odot_{\Sigma' \setminus n}$ is used instead of $\odot_{\Sigma'}$, where n is the name of the spawned thread.*
3. $\acute{\Sigma} = \Sigma + \Sigma'$

We write Φ+a for the update. The update for outgoing communication is defined dually in the sense that $\odot_n$ of a spawn label is added to Δ instead of Θ. Likewise, the $\odot_\Sigma$ (resp. $\odot_{\Sigma' \setminus n}$) are added to Θ, instead of Δ.

Now to the update of *connectivity*. We concentrate again on incoming communication; the situation for outgoing communication is dual. The general intention of updating environment connectivity is clear: incoming communication may bring entities in connection which had been separate before, in particular it may merge object cliques. For the commitment context, this can be directly formulated by adding the fact that the receiver of the communication now is acquainted with all transmitted arguments. See part (1) of Definition 6 below. For the update of *assumption* connectivity context E_Δ, we add that the sender knows all of the names which are transmitted boundedly (cf. part (2) of Definition 6). No update occurs wrt. names already known.

Note that the sender of a communication may itself not be contained in Δ before the communication: This situation occurs only for call and spawn steps, more precisely for incoming spawn steps and incoming calls, where the calling thread enters the component for the first time; for incoming returns, the sender is already known (and determined). Indeed, for an incoming call or spawn, the sender may not only be unknown, i.e., not mentioned in Δ before the step, it may remain anonymous after, as well. Furthermore, even if it's clear that the communication must originate from the environment, there can be more than one possible environment cliques as source, when the thread is new. In the operational rules, the update of Definition 6 is used where the sender is appropriately guessed in those circumstances.

Remains the treatment of *thread names* transmitted boundedly. Let us first assume that they do not include the active thread. As mentioned, for each thread n', the contexts remember where the thread starts its life, using the symbol $\odot_{n'}$ to denote the "initial clique" of thread n. The initial clique may not contain real objects, namely if the thread is instantiated without handing over object identities via the thread constructor. The semantics maintains as invariant that for each thread name n mentioned in the Σ-context, either $\Delta \vdash \odot_n$ or $\Theta \vdash \odot_n$: A thread known both at the environment and the components started on exactly one side. The thread exchanged in Σ' have not yet crossed the border actively (indeed their names have not even passed the border in argument position, for that matter). It it clear, however, if they start being active at the interface, if ever, their first interaction will be an *incoming* call. To remember this circumstance, $\odot_{n'}$ for all thread identities from Σ' (abbreviated $\odot_{\Sigma'}$) is added to the environment context. Furthermore we may assume that they belong to the clique of the sender, which we fix by adding $o_s \hookrightarrow \odot_{\Sigma'}$ to the connectivity assumptions.

Definition 6 (Connectivity context update). *The update $(\acute{E}_\Delta, \acute{E}_\Theta)$ of an assumption-commitment context (E_Δ, E_Θ) wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor?$ with sender o_s and receiver o_r is defined as follows.*

1. $\acute{E}_\Theta = E_\Theta + o_r \hookrightarrow \lfloor a \rfloor$.
2. $\acute{E}_\Delta = E_\Delta + o_s \hookrightarrow \Phi', \odot_{\Sigma'}$. *In case of a spawn label, $\odot_{\Sigma' \setminus n}$ is used instead of $\odot_{\Sigma'}$, where n is the name of the spawned thread.*

We write $(E_\Delta, E_\Theta) + o_s \xrightarrow{a} o_r$ for the update.

Combining Definitions 5 and 6, we write $\Xi + o_s \xrightarrow{a} o_r$ when updating the name and the connectivity at the same time.

Besides Definition 4, which checks whether a label is possible in that the connectivity assumptions are met, we must not forget the *static* assumptions, i.e., whether the transmitted values are of the correct types. This is covered in the following definition.

Definition 7 (Well-typedness of a label). Well-typedness *of an outgoing label a relative to the contexts Δ, Σ, Θ is given by the rules of Table 8. We use $\Delta, \Sigma, \Theta \vdash a : ok$ as notation to assert well-typedness.*

$$\frac{\Phi' = \Delta', \Sigma', \Theta' \quad dom(\Phi') \subseteq fn(n\langle call\ o_r.l(\vec{v})\rangle) \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} = \Delta, \Sigma, \Theta + \Phi' \quad ; \acute{\Theta} \vdash o_r{:}c_r \quad ; \Delta, \Theta \vdash c_r : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad ; \Delta', \Sigma', \Theta' \vdash \vec{v} : \vec{T}}{\Delta, \Sigma, \Theta \vdash \nu(\Phi').n\langle call\ o_r.l(\vec{v})\rangle?\ {:}ok}$$

$$\frac{\Phi' = \Delta', \Sigma', \Theta' \quad dom(\Phi') \subseteq \langle spawn\ n\ of\ c_t(\vec{v})\rangle? \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} = \Delta, \Sigma, \Theta + \Phi' \quad \Delta, \Theta \vdash c_t : \vec{T} \to thread \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \vec{v} : \vec{T}}{\Delta, \Sigma, \Theta \vdash \nu(\Phi').\langle spawn\ n\ of\ c_t(\vec{v})\rangle?\ {:}ok}$$

$$\frac{\Phi' = \Delta', \Sigma', \Theta' \quad dom(\Phi') \subseteq n\langle return(v)\rangle? \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} = \Delta, \Sigma, \Theta + \acute{\Phi} \quad ; \Delta, \Theta \vdash c_r : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad ; \Delta, \Sigma, \Theta \vdash v : T}{\Delta, \Sigma, \Theta \vdash \nu(\Phi').n\langle return(v)\rangle?\ {:}ok}$$

Table 8. Checking static assumptions

Remark 5 (Without thread classes). In a setting without thread classes and where thread names cannot be sent around in the programming language, still thread names are exchanged by scope extrusion: When a new thread crosses actively the border, its name occurs in subject position in the label and is transmitted boundedly.

Without thread classes, however, the setting simplifies in that one does not need $\odot_n$. We introduced $\odot_n$ as a representation of the clique where n started its life, n's *urvater.* This "virtual object" is needed for the situation, when a thread name crosses the component-environment border for the first time, i.e., in the situation of a *first* cross-border method call of a thread.

For example in an incoming call and without thread classes, it is clear that the new thread has started its life in the environment; hence the semantics needs no representation of that fact, for instance by requiring $\Delta \vdash \odot_n$ as here. Furthermore, one additional fact is guaranteed: the thread is spawned by some *existing* environment clique mentioned in Δ.[6] In that setting, the spawner of the new thread still needs to be guessed, but the semantics can pick a spawner from the objects already known.

With thread instantiation, this assumption is no longer valid: the component can create an environment thread without connection to any environment object, or even without connection to any object. If such a thread then calls back (or it creates internally another thread that then enters the component), the semantics must check whether it is possible that it knows all of the values mentioned in the label.

In short, the presence of thread classes requires the introduction of $\odot_n$ into the semantics. □

Remark 6 (Without communication of thread names). In [3], we considered a setting which is restricted not only in that it does *not* feature thread classes but

[6] The only exception is the very first step.

also in that *communication* of thread names is not supported. More precisely, the only way in that thread names are communicated is in subject position of the label. The thread name in this situation is still part of the communication label, as the expression *currentthread* allows to observe the name; in particular, the setting of [3] allows to store thread identities, it is only not possible to mention the name as argument in method calls.

The fact that the thread name in subject position is observable by the callee of a method in some sense means that the mentioned restriction not to communicate thread in argument position is not a real restriction. On the other hand, since thread names do not appear in communication labels in object position, there is no need to *check* whether this name may be known by the respective caller. As a consequence, the setting is simplified in that the connectivity of objects and threads, i.e., pairs of the form $o \hookrightarrow n$ are *not* needed for a fully abstract semantics. □

Operational rules With all the ancillary definitions at hand, we can define the operational rules of the semantics (cf. Table 9).

The three CallI-rules deal with incoming calls. For all three cases, the contexts are *updated* to $\acute{\Xi}$ to include the information concerning new objects, threads, and connectivity transmitted in that step. Furthermore, it is *checked* whether the label statically type-checks and that the step is possible according to the (updated) connectivity assumptions $\acute{\Xi}$. Remember that the update from Ξ to $\acute{\Xi}$ includes guessing of connectivity, i.e., an element of non-determinism, when the sender of the communication is unknown to the component.

The three rules for incoming calls deal with three different situations as to when an incoming call may happen: A reentrant call[7], a call of thread where the thread name is already known in the component, and a call of a thread which is new to the component.

To deal with component entities (threads and objects) that are being created during the call $C(\Theta', \Sigma')$ stands for $C(\Theta') \parallel C(\Sigma')$, were $C(\Theta')$ are the lazily instantiated objects mentioned in Θ'. Furthermore, for each thread name n' in Σ', a new component $n'\langle stop \rangle$ is included, written as $C(\Sigma')$.

The treatment of the connectivity contexts is uniform in all three cases, only the identity of the sender is different.

For reentrant method calls (cf. rule $\textsc{CalliI}_1$), the thread is blocked, i.e., it has left the component previously via an outgoing call. The object that had been the target of the call is remembered as part of the augmented block syntax. In the rule it is referred to as o_s, as it is the sender of the current incoming call. Two points are worth mentioning: first, o_s needs not be the *actual* caller, which remains anonymous, since the callee cannot observe who really calls. The reference o_s, however, can be taken as representative of the environment clique from which the call is being issued: the call must originate from the clique where it has previously left into since it cannot enter a disjoint environment clique, at least not without detour via the component which would have been observable

[7] Reentrant on the level of the component, not on the level of a single object.

and recorded in the connectivity contexts. Secondly, note that the object o_s stored in the block-syntax is not necessarily the callee of the call the thread did *immediately prior* to this incoming call. In the history of the thread, there might have been message exchange in between the blocked outgoing call and the current incoming call, whose code has been popped off the stack. Nonetheless, o_s must (still) be in the clique which sends the current call.

Rule CALLI$_2$ treats a non-reentrancy situation, where the thread name is already known in the component nonetheless. As a consequence, the component contains the entity $n\langle stop\rangle$. Unlike in rule CALLI$_1$, the program code contains no indication as to the origin of the call. Since the thread n must have crossed the border before, the marker for its initial clique $\odot_n$ must be contained in either Δ or in Θ. The premise $\Delta \vdash \odot_n$ assures that n had started its life on the environment side. This bit of information is important as otherwise one could mistake the code $n\langle stop\rangle$ for the code of a (deadlocked) outgoing call. If $\Delta \vdash \odot_n$ and $n\langle stop\rangle$ is part of the component code, it is assured that the thread either has never actively entered the component before (and does so right now) or has left the component to the environment by some last outgoing return. In either case, the incoming call is possible now, and in both cases we can use $\odot_n$ as representative of the caller's identity.

The last call rule CALLI$_3$ deals with the situation, that the thread n enters the component for the first time. This is assured by the premise $\Sigma' \vdash n : thread$. As in CALLI$_2$, we do not have an indication from which clique the call originates, since the corresponding thread is *new.*

What is assured is that the new thread has been created at some point before as instance of some environment thread class —otherwise the cross-border instantiation would have been observed and the thread name would not be fresh now— and by some environment clique. Indeed, *any* existing environment clique is a candidate that might have created the thread n. So the update to $\acute{\Xi}$ *non-deterministically guesses* to which environment clique the thread's origin $\odot_n$ belongs to. Note that $\odot_{\Sigma'}$ contains $\odot_n$ since $\Sigma' \vdash n$, which means $\acute{\Delta} \vdash \odot_n$ after the call.

For incoming thread creation in rule SPAWNI, we need again to know the origin of the call, i.e., the spawning clique. The situation is similar to the one for CALLI$_3$, in that the origin of the communication needs to be guessed. In the case of CALLI$_3$, we use $\odot_n$ as "virtual clique", i.e., as representative for the calling clique, covering the situation where no actual calling object may be the source. Different from the situation of unknown caller is that here we can obviously not use $\odot_n$; that identity is incorporated into the *component* after the call. What is clear is that the spawner must be part of the environment prior to the call, i.e., $\Delta \vdash o_s$, where o_s might be some $\odot_{n'}$, i.e., a virtual clique of objects from which no actually existing objects has yet escaped to the component. Note that if $o_s = \odot_{n'}$, $\Delta \vdash o_s$ assures that $n \neq n'$. Note further that the name of the spawned thread is treated specifically in the definition of context update (cf. Definitions 5 and 6) to cater for cross-border instantiation of the new thread.

$$\frac{\begin{array}{c}\acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} o_r : ok \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \lfloor a \rfloor : ok \quad \acute{\Theta} \vdash o_r \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad t_{blocked} = let\ x{:}T' = o\ blocks\ for\ o_s\ in\ t\end{array}}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle t_{blocked}\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\Phi).(C \parallel C(\Theta', \Sigma') \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ return\ to\ o_s\ x; t_{blocked}\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{CalliI}_1$$

$$\frac{\begin{array}{c}\acute{\Xi} = \Xi + \odot_n \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \overset{\lfloor a \rfloor}{\rightarrow} o_r \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \lfloor a \rfloor\ ok \quad \acute{\Theta} \vdash o_r \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \Delta \vdash \odot_n\end{array}}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash C \parallel n\langle stop\rangle : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma') \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ return\ to\ \odot_n\ x; stop\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{CalliI}_2$$

$$\frac{\begin{array}{c}\acute{\Xi} = \Xi + o \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \overset{\lfloor a \rfloor}{\rightarrow} o_r \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \lfloor a \rfloor : ok \quad \acute{\Theta} \vdash o_r \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \Delta \vdash o \quad \Sigma' \vdash n : thread\end{array}}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ return\ to\ \odot_n\ x; stop\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{CalliI}_3$$

$$\frac{\begin{array}{c}a = \nu(\Phi').\langle spawn\ n\ of\ c_t(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_n \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} \odot_n \quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \lfloor a \rfloor : ok \\ \acute{\Theta} \vdash o_r \quad \Delta \vdash o_s \quad \Phi' = \Theta', \Delta', \Sigma' \quad \Theta \vdash c_t \quad \Sigma' \vdash n : thread\end{array}}{\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle c_t(\vec{v})\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta}\ \textsc{SpawnI}$$

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle! \quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi \quad \acute{\Phi} = \Phi \setminus \Phi' \quad \acute{\Delta} \vdash o_r \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ o_r.l(\vec{v})\ in\ t\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = o_s\ blocks\ for\ o_r\ in\ t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{CallO}$$

$$\frac{\begin{array}{c}a = \nu(\Phi').\langle spawn\ n'\ of\ c_t(\vec{v})\rangle! \quad \Phi' = (fn(\lfloor a \rfloor) \setminus n') \cap \Phi \quad \acute{\Phi} = \Phi \setminus \Phi' \\ \Delta \vdash c_t \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_{n'}\end{array}}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ spawn\ c_t(\vec{v})\ in\ t\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = n'\ in\ t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{SpawnO}$$

Table 9. External steps

An incoming spawn action without known external objects is possible only in the very first step and is covered by $\textsc{SpawnI}_0$ from Table 11.

Outgoing calls are dealt with in rule CallO. To distinguish the situation from component-internal calls, the receiver must be part of the environment, which is expressed by $\acute{\Delta} \vdash o_r$. Note that the identity o_r may be contained in the bound names Δ' of the label, i.e., the callee o_r may be lazily instantiated by the outgoing call. The connectivity assumption contexts are updated by the information that the callee may now know the thread name and all arguments. For the commitment context, we must add connectivity information concerning the names whose scope now extrudes to the environment.

The sender o_s is contained in the code as part of the augmentation, so no guessing is involved this time. Outgoing communication is simpler also wrt. static

type checking: Assuming that we start with a well-typed component, there is no need in re-checking now that only values of appropriate types are handed out, since the operational steps preserve well-typedness ("subject reduction").

The boundedly transmitted thread names Σ' now contains the threads instantiated from component thread classes and whose life starts at the component side. We simply extend the commitments by the additional information that they belong to the sender's clique by adding $o_s \hookrightarrow \odot_{\Sigma'}$.

For outgoing thread creation (cf. rule SpawnO), the action updates the assumption context in the following manner. The name context Δ is extended by the environment names transmitted boundedly, which in particular includes the name of the new thread. In addition we must remember which references are handed over to the new thread to detect situations, when the thread later calls back with references it cannot possibly know (cf. also Remark 8). As before, $\odot_{n'}$ denotes the initial clique of environment objects the thread starts in, which is in acquaintance with the arguments $\vec{v}$ after the step. The thread names transmitted in subject position in Σ', which refer to threads that start in the component, are treated as in CallO, where o_s in the augmented code represents the spawning clique. Unlike the treatment of the outgoing call, o_s needs not be remembered in the code, as the thread never returns.

The rules of Table 10 deal with the return actions and lazy instantiation of objects. The return steps work similar as the calls. They are simpler, however, since the element of guessing is not present: when a thread returns, the callee as well as the thread are already known. Returns are simpler than calls also in that only one value is communicated, not a tuple (and we don't have compound types). To avoid case distinctions and to stress the parallel with the treatment of the calls, we denote the binding part of the label by $\nu(\Phi')$ resp. $\nu(\Delta', \Sigma', \Theta')$ as before, even if at least two of the name contexts are guaranteed to be empty. Rule NewO_{lazy} deals with lazy instantiation and describes the local instantiation of an external class. Instead of exporting the newly created name of the object plus the object itself immediately to the environment, the name is kept local until, if ever, it gets into contact with the environment. When this happens, the new instance will not only become known to the environment, but the object will also be instantiated in the environment. Note that the instantiation is a confluent step. Nevertheless, it is part of the external semantics in that it references the assumption context.

The initial steps are axiomatized in Table 11. Obviously, initially no returns are possible. The rules are variants of the rules from Table 9, where it is required that the assumption and commitment contexts do not contain object or thread names, formalized as $\Delta_0, \Theta_0 \vdash \mathit{static}$. There is exactly one initial thread, either in the component or in the environment. Where the initial activity starts is marked by $\odot$. For the initial static contexts, we are given either $\Delta_0 \vdash \odot$ or $\Theta_0 \vdash \odot$. Note that in rules CallO_0 and SpawnO_0, the sender needs not be the initial clique. The first outgoing environment interaction is not necessarily caused by the initial code fragment; the component might start with internal method calls,

$$\frac{a = \nu(\Phi').\ n\langle return(v)\rangle?\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r\quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \lfloor a \rfloor : ok\quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : ok}{\begin{array}{c}\Delta; E_\Delta \vdash C \parallel n\langle let\ x{:}T = o_r\ blocks\ for\ o_s\ in\ t\rangle : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel n\langle t[v/x]\rangle : \acute{\Theta}; \acute{E}_\Theta\end{array}}\ \textsc{RetI}$$

$$\frac{a = \nu(\Phi').\ n\langle return(v)\rangle!\quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi\quad \acute{\Phi} = \Phi \setminus \Phi'\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ return\ to\ o_r\ v\ in\ t\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{RetO}$$

$$\frac{\Delta \vdash c}{\begin{array}{l}\Delta, \Sigma; E_\Delta \vdash n\langle let\ x{:}c = new\ c\ in\ t\rangle : \Theta, \Sigma; E_\Theta \rightsquigarrow \\ \qquad \Delta, \Sigma; E_\Delta \vdash \nu(o{:}c).n\langle let\ x{:}c = o\ in\ t\rangle : \Theta, \Sigma; E_\Theta\end{array}}\ \textsc{NewO}_{lazy}$$

Table 10. External steps (2)

and indeed the active thread as the subject of the interaction need not be the initial thread.

$$\frac{\begin{array}{c}\Delta_0, \Theta_0 \vdash static\quad \Delta_0 \vdash \odot\quad \acute{\Xi} = \Xi + \odot_n \xrightarrow{a} o_r\quad \acute{\Xi} \vdash \odot_n \xrightarrow{\lfloor a \rfloor} o_r : ok\quad \Delta_0, \Sigma_0, \Theta_0 \vdash a : ok \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle?\quad \acute{\Sigma} \vdash n : thread\end{array}}{\begin{array}{l}\Delta_0 \vdash C : \Theta_0 \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ return\ to\ \odot\ x; stop\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{CalliI}_0$$

$$\frac{\begin{array}{c}\Delta_0, \Theta_0 \vdash static\quad \Theta_0 \vdash \odot\quad a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle! \\ \Phi' = fn(\lfloor a \rfloor) \cap \Phi\quad \acute{\Phi} = \Phi \setminus \Phi'\quad \acute{\Delta} \vdash o_r\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r\end{array}}{\begin{array}{l}\Delta_0 \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ o_r.l(\vec{v})\ in\ t\rangle) : \Theta_0 \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = o_s\ blocks\ for\ o_r\ in\ t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{CalliO}_0$$

$$\frac{\begin{array}{c}\Delta_0, \Theta_0 \vdash static\quad \Delta_0 \vdash \odot\quad \acute{\Xi} = \Xi + \odot \xrightarrow{a} \odot_n\quad \acute{\Xi} \vdash \odot \xrightarrow{\lfloor a \rfloor} \odot_n\quad \acute{\Theta}, \acute{\Sigma}, \acute{\Delta} \vdash \lfloor a \rfloor : ok \\ a = \nu(\Phi').\langle spawn\ n\ of\ c_t(\vec{v})\rangle?\quad \Theta \vdash c_t\quad \Sigma' \vdash n : thread\end{array}}{\Delta_0 \vdash C : \Theta_0 \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle c_t(\vec{v})\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta}\ \textsc{SpawniI}_0$$

$$\frac{\begin{array}{c}\Delta_0, \Theta_0 \vdash static\quad \Theta_0 \vdash \odot\quad a = \nu(\Phi').\langle spawn\ n'\ of\ c_t(\vec{v})\rangle! \\ \Phi' = (fn(\lfloor a \rfloor) \setminus n') \cap \Phi\quad \acute{\Phi} = \Phi \setminus \Phi'\quad \Delta \vdash c_t\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_{n'}\end{array}}{\begin{array}{l}\Delta_0 \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_s\ \ spawn\ c_t(\vec{v})\ in\ t\rangle) : \Theta_0 \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = n'\ in\ t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta\end{array}}\ \textsc{SpawnO}_0$$

Table 11. Initial external steps

Remark 7 (Hiding, thread classes and types). Note the following (rather arcane) point in connection with new threads entering actively the component, i.e., in connection with rule CALLI$_2$. The fact that the language allows to *hide components* via the ν-binder assures that, as we claimed, one of the existing environment cliques can indeed be the origin of the new thread! If all thread classes would be visible at the interface, then there might not be any thread class available which we could hand over object references to the new thread in such a way that it can call back to the component. This failure would ultimately compromise the completeness of the semantics.

Without the ability to hide (thread) classes, a new thread in the above situation can be invisibly created by an existing environment clique appropriately only under the following assumption: there must be at least one environment clique with a thread constructor where at least one argument is of an *environment* class type. The constructor is needed to hand over to the new thread at least one object reference which can be used to distribute connectivity information. The object used for that needs to be environment object; otherwise the interaction with the object would be visible. Note also that if it is possible for an environment clique to spawn the new thread, *any* existing environment clique can spawn it. Indeed the only situation under these circumstances, where the spawning is not possible then is when there is no environment clique *at all* and where the main thread has started in the component, i.e., at the very beginning.

Related to the discussion: it is important that thread names are typed by *thread*, and not by the *name* of their thread class (cf. also Remark 2). If the type were the thread class, then the semantics would have to *forbid* that the instances of hidden thread classes cross the border between environment and component, otherwise subject reduction would break. So this restriction would not solve the above mentioned problem. The alternative would be that scope extrusion of a thread name would entail scope extrusion of the name of the thread class, which would guarantee subject reduction, but this "solution" seems strongly unplausible namely: getting hold of a thread name gives the power to instantiate the corresponding thread class. For objects, which are typed by the name of their class, the problem is not present as the caller in a method call remains anonymous. □

Remark 8 (Thread constructors). The spawner can hand over values to the new thread via the thread constructor. For ordinary class instantiation, where we don't have constructors, we stated that the absence of constructors entailed lazy instantiation. Note, however, that in the case of thread instantiation, we would have eager instantiation even if we disallowed constructors. The reason is, that upon instantiation, the new thread is active from the beginning.

In case of threads, however, the absence of constructors would be more drastic: without acquaintance with objects handed over at instantiation time, the new thread would not be able to contact any of the existing objects in the component as well as in the environment. The spawner is "acquainted" with the new thread in that it knows its identity but it cannot "communicate" with its child thread. In some sense, the only point where the spawner can communicate with its child

thread is during instantiation, and without this possibility, the multithreading would degenerate to a program consisting of groups each with one single threads which are *globally* completely separate, i.e., not simply separate when considered the connectivity as seen from the component or the environment. Note that it does not mean that for instance an environment thread created by the component via SpawnO cannot "call back" to the component, only that for calling back it needs to create its own cliques of objects unrelated to the rest. □

3 Conclusion

Smith [15] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* [16][13] specification language. called the *complete-readiness* model, related to the readiness model of Olderog and Hoare [12]. [17] investigates full abstraction in an object calculus with subtyping. The setting is a bit different from the one as used here as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. Recently, Jeffrey and Rathke [11] extended their work on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*. Cf. also [14]. [6] tackles the problem of full abstraction and observable component behavior and connectivity in a UML-setting.

Future work The trace semantics together with the equivalence relation capturing the undefinedness of order of interacting with separate cliques is a "*tree*" semantics. The semantics can be understood as a forest of interactions, where each tree represents one current clique of objects. The cliques can be dynamically created and the branching structure is caused by merging of cliques. We are currently working on a *direct* tree representation of the semantics. The resulting semantics is simpler as it can do without the secondary notion of equivalence relation on traces, and furthermore one can avoid an explicit representation of object connectivity. However, e.g., the derivation system for legal traces gets more involved in that it must reflect the branching structure.

Acknowledgements We thank Harald Fecher, and Marcel Kyas, and Willem-Paul de Roever for stimulating discussions on various aspects of this topic.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Aug. 2003.

3. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Z. Li, editor, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, July 2004.
4. E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In Bosangue et al. [5]. To appear.
5. M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors. *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
6. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [5]. To appear.
7. ECMA International Standardizing Information and Communication Systems. *C# Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.
8. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
9. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.
10. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.
11. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. 2005. Submitted for publication.
12. E.-R. Olderog and C. A. R. Hoare. Specification-oriented semantics of communicating processes. *Acta Informatica*, 23(1):9–66, 1986. A preliminary version appeared under the same title in the proceedings of the 10th ICALP 1983, volume 154 of LNCS.
13. B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Series in Computer Science. Prentice Hall, 1990.
14. J. Rathke. A fully abstract trace semantics for a core Java language (preliminary title). In Bosangue et al. [5]. To appear.
15. G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
16. J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 1989.
17. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.

Index

List of Tables