INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Heap-Abstraction for an Object-Oriented Calculus with Thread Classes

Erika Ábrahám Andreas Grüner
Martin Steffen

Bericht Nr. 0601
February 1, 2006

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

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

Heap-Abstraction for an Object-Oriented Calculus with Thread Classes

Erika Ábrahám Andreas Grüner
Martin Steffen

Bericht Nr. 0601
February 1, 2006

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

Part of this work has been financially supported by the NWO/DFG project Mobi-J (RO 1122/9-4) and by the DFG project AVACS (SFB/TR-14-AVACS).

Heap-Abstraction for an Object-Oriented Calculus with Thread Classes

February 1, 2006

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

[1] Albert-Ludwigs-University Freiburg, Germany, eab@informatik.uni-freiburg.de
[2] Christian-Albrechts-University Kiel, Germany, {ang,ms}@informatik.uni-kiel.de

Abstract. This paper formalizes an open semantics for a calculus featuring thread classes, where the environment, consisting in particular of an overapproximation of the heap topology, is abstractly represented. 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 to account explicitly for the possible acquaintance of objects with threads. We show soundness of the abstraction.

Keywords: class-based oo languages, thread-based concurrency, open systems, formal semantics, heap abstraction, observable behavior

Table of Contents

1 Introduction 2
2 A multi-threaded calculus with thread classes 4
2.1 Syntax 4
2.2 Type system 5
2.3 Operational semantics 7
2.3.1 Internal steps 8
2.3.2 External steps 10
2.3.3 Connectivity contexts 11
2.3.4 Augmentation 13
2.3.5 Use and change of contexts 14
2.3.6 Operational rules 16
3 Legal traces 20
3.1 Linear representation 21
3.1.1 Balance conditions 22
3.1.2 Legal traces system 24
3.2 Branching system 26
3.3 Equivalence of the legal traces representations 30
3.4 Soundness of the abstractions 35
4 Conclusion 36
References 37

1 Introduction

An *open* system is a program fragment or component interacting with its environment or context. In a message-passing setting, the behavior of the component can be understood to consist of message traces at the interface, i.e., of sequences of component-environment interaction. Even if the environment is absent, it must be assured that the component together with the (abstracted) environment gives a well-formed program adhering to the syntactical and the context-sensitive restrictions of the language at hand. Technically, for an exact representation of the interface behavior, the semantics of the open program needs to be formulated under *assumptions* about the environment, capturing those restrictions. The resulting assumption-commitment framework gives insight to the semantical nature of the language. Furthermore, an independent characterization of possible interface behavior with environment and component abstracted can be seen as a trace logic under the most general assumptions, namely conformance to the inherent restrictions of the language and its semantics.

With these goals in mind, we deal primarily with the following three features, which correspond to those of modern class-based object-oriented languages like *Java* [12] or $C^{\#}$ [10] and which are notoriously hard to capture:

- *types and classes:* the languages are statically typed and only well-typed programs are considered. For class-based languages, complications arise as classes play the role of types and additionally act as *generators* of objects.
- *concurrency:* the languages feature concurrency based on *threads* and *thread classes* (as opposed to processes or active objects [7]).
- *references:* each object carries a unique *identity.* New objects are dynamically allocated on the heap as *instances of classes.*

We investigate the issues in a class-based multi-threaded calculus with thread classes. The interface behavior is phrased in an assumption-commitment framework and is based on three orthogonal abstractions:

- a static abstraction, i.e., the type system;
- an abstraction of the stacks of recursive method invocations, representing the recursive and reentrant nature of method calls in a multi-threaded setting;
- an abstraction of the *heap topology,* approximating potential connectivity of objects and threads. The heap topology is dynamic in that new objects may be created and tree-structured in that previously separate object groups may merge.

In [3,4] we showed that the need to represent the heap topology is a direct consequence of considering *classes* as a language concept. Their foremost role in object-oriented languages is to act as *"generators of state"*. With *thread classes,* there is also a mechanism for *"generating new activity"*, i.e., for creating new threads. 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.

We concentrate on the third point, the abstraction of the heap topology. 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* [12] and $C^{\#}$ [10], 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". Without thread 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.

Thus, the technical contribution of this paper is threefold. We extend the class-based calculus and its semantics of [3,4] to include *thread classes* and furthermore allow the communication of thread names. This requires to consider

cross-border *activity* generation as well as to incorporate the connectivity of objects *and* threads. Secondly, we characterize the potential traces of *any* component in an assumption-commitment framework in a novel derivation system, where the branching nature of the heap abstraction —connected groups of objects can merge by communication— is reflected in the branching structure of the derivation system. Finally, we show the soundness of the mentioned abstractions.

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 contains an independent characterization of the observable behavior of an open system and the soundness results of the abstractions. Section 4 concludes with related and future work. For a full account of the operational semantics and the type system, we refer to the technical report [6].

2 A multi-threaded calculus with thread classes

Next we present the calculus, starting with the syntax. It is based on the multi-threaded object calculus, similar to the one presented in [11] and in particular [13]. 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 class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields. A method $\varsigma(\mathit{self}{:}c).t_a$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. We use $\varsigma \mathit{self}{:}c\bot_c$ for the undefined value of a field. An object $o[c, F]$ keeps a reference to the class it instantiates and stores the current values of the fields or instance variables.

Thread classes $c_t\langle\!\langle t_a \rangle\!\rangle$ are known under the name c_t and carry their abstract 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 like in Table 1.

Besides named objects and classes, the dynamic configuration of a program contains threads $n\langle t \rangle$ as active entities. A thread is basically either a value or a sequence of expressions, notably method calls $v.l(\vec{v})$, the creation of new instances $\mathit{new}\, c$ of a class c, and *thread instantiation* $\mathit{spawn}\, c_t(\vec{v})$. We use l for instance variables or fields, $l = f$ for field variable declaration, field access is written as $v.l$, and field update[3]as $v.l := v'$. A field can be can be tested for undefinedness by *undef* in the second conditional expression.

The available types are given in the grammar of Table 2, where we use $\mathsf{Unit} \to T$ for $T_1 \times \ldots T_n \to T$, when $n = 0$.

[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	$::= \varsigma(n{:}T).t_a$	method
f	$::= \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_c$	field
t_a	$::= \lambda(x{:}T, \ldots, x{:}T).t$	parameter abstraction
t	$::= v \mid \mathit{stop} \mid \mathit{let}\, x{:}T = e\, \mathit{in}\, t$	thread
e	$::= t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}\,(v.l)\, \mathsf{then}\, e\, \mathsf{else}\, e$	expression
	$\mid\ v.l(v, \ldots, v) \mid v.l := v$	
	$\mid\ \mathit{currentthread} \mid \mathit{new}\, n \mid \mathit{spawn}\, n(v, \ldots, v)$	
v	$::= x \mid n$	values

Table 1. Abstract syntax

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 $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$.

2.2 Type system

The type system or static semantics presented next characterizes the well-typed programs. The derivation rules are shown in Tables 3 and 4.

Table 3 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

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

Table 2. Types

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 names inside the component (cf. rule T-Nu$_i$ for the internal and rule T-Nu$_e$ for the external case). All names can be hidden, i.e., class names, in particular names of thread classes, as well as object and thread references. Since class names 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 for class names and T-NTClass for thread class names); 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 in rule T-NThread, 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]

Subsumption from rule T-Sub expresses a simple form of subtyping: we allow that an object respectively class contains *at least* the members that 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 4 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 [13] 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

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

[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.

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

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

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

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

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

$$\frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-Sub}$$

Table 3. Static semantics (components)

The external steps, presented in Section 2.3.2, defines the interaction between component and environment and is given in reference to assumption-commitment contexts. The static part of the context corresponds to the static type system from Section 2.2 on the component level and takes care that, e.g., only well-typed values are received from the environment. The context, however, needs to contain also a *dynamic* part dealing with the potential *connectivity* of objects and thread names, corresponding to an abstraction of the heap topology, as discussed in Section 2.3.3.

2.3.1 Internal steps For the internal steps of Table 5, 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 for one single thread. 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

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots, l{:}\mathsf{Unit} \to T, \ldots]\!] \quad \Gamma;\Delta \vdash e_1 : T_2 \quad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ undef(v.l())\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-Cond}_{\perp}$$

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

Table 4. Static semantics (threads and objects)

as template for the created object, and the identity of the object is new and 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 Spawn_i specifies internal thread class instantiation.

Rule 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 $t[o/self][\vec{v}/\vec{x}]$, when the method suite $[M]$ equals $[\ldots, l = \varsigma(self{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Note also

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

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

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

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

$n\langle let\, x{:}T = (\mathsf{if}\, undef\,(\varsigma(s{:}c)\lambda().\bot_{c'})\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, in\, t\rangle \rightsquigarrow$
$n\langle let\, x{:}T = e_1\, in\, t\rangle$ $\textsc{Cond}_1^{\bot}$

$n\langle let\, x{:}T = (\mathsf{if}\, undef\,(\varsigma(s{:}c)\lambda().v)\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2)\, in\, t\rangle \rightsquigarrow$
$n\langle let\, x{:}T = e_1\, in\, t\rangle$ $\textsc{Cond}_2^{\bot}$

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

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

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

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

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

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

Table 5. Internal steps

that the step is a $\xrightarrow{\tau}$-step, not a confluent one. The same holds for field update in rule FUPDATE, where $[c, (l_1 = f_1, \ldots, l_k = f_k, l = v').l := v]$ stands for $[c, l_1 = f_1, \ldots, l_k = f_k, l = v]$. Note further that instances of a component class invariantly belong to the component and not to the environment. I.e., an instance of a component class resides in the component, and named objects are never exported from the component to the environment or vice versa; of course, *names* to objects may well be exported.

The reduction relations are used modulo *structural congruence* $\equiv$, which captures the algebraic properties of parallel composition and hiding. The basic axioms for $\equiv$ are shown in Table 6 where in the fourth axiom, n does not occur free in C_1. The congruence relation is imported into the reduction relations in Table 7. Note that all syntactic entities are always tacitly understood modulo α-conversion.

2.3.2 External steps A component exchanges information with the environment via *call, return,* and *spawn* actions (cf. Table 8). In call and return labels,

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

Table 6. Structural congruence

n is the active thread that issues the call or returns from the call. For thread instantiation, n is the new thread; the spawning 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. There are no labels for object creation: Externally instantiated objects are created only at the point when they are accessed for the first time, which we call "*lazy instantiation*". For a label $\nu(\Phi).\gamma?$ or $\nu(\Phi).\gamma!$ 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 where γ does not contain any binders, we call γ the *core* of the label. The core of label a we denote by $\lfloor a \rfloor$.

2.3.3 Connectivity contexts With cross-border instantiation, the semantics must contain a representation of the connectivity, which is formalized by a relation on the names of the calculus and which can be seen as an abstraction of the program's heap; see equations (2) and (3) below for the exact definition. 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

[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 7. 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 8. Labels

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. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names. 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. 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.

The semantics must book-keep which objects of the environment have been told which identities: It takes into account the *relation* of objects from the assumption context Δ amongst each other, and the knowledge of objects from Δ about thread names and names exported by the component. In analogy to the name contexts Δ and Θ, the connectivity context E_Δ expresses assumptions about the environment, and E_Θ commitments of the component:

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

Since thread names may be communicated, we 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 the relations E_Δ and E_Θ. 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:

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

where $\hookrightarrow\!\downarrow_\Delta$ is the projection of $\hookrightarrow$ to Δ. 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 dual.

The relation $\leftrightharpoons$ partitions the objects from Δ (resp. Θ) 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.

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 updated in incoming communications, while the assumption context E_Δ is consulted to exclude impossible incoming values, and is updated in outgoing communications. Incoming new names, exchanged boundedly, however, update both commitments and assumptions.

Remark 2 (Initial 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. For appropriately dealing with the connectivity in those cases, we need a syntactical representation for the clique of objects, the thread n starts in; we use the symbol $\odot_n$ as n's *initial clique.* As said, the symbol $\odot_n$ 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.

Concerning $\odot_n$, the semantics maintains as invariant that a thread name n occurs in the context Σ for thread names, iff. $\odot_n$ occurs in either Δ or Θ, the contexts containing the objects (plus class definitions). This means, besides being relevant for connectivity information, $\odot_n$ contains also the information whether the thread started its life in the environment or in the component.

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 (potentially $\odot_n$) is clear. □

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

$$o_1 \ \mathit{blocks\ for}\ o_2 \quad \text{and} \quad o_2 \ \mathit{returns\ to}\ o_1\ v\ ,$$

denoting a method body in o_1 waiting for a return from o_2, and dually for the return of v from o_2 to o_1. We augment the method definitions accordingly, such that each method call and spawn action is annotated by the caller. I.e., we write

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

instead of $\varsigma(\mathit{self}{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y}) \ldots \mathit{spawn}\, c_t(\vec{z}) \ldots)$. The code in the thread classes is augmented by the thread's initial clique as follows:

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

If a thread n is instantiated, $\odot$ is replaced by $\odot_n$. For a thread class of the form $c_t\langle\!\langle \lambda(\vec{x}{:}\vec{T}).t\rangle\!\rangle$, let $c_t(\vec{v})$ denote the replacement $t[\odot_n, \vec{v}/\odot, \vec{x}]$. After instantiation,

the thread class looks as follows: $n\langle \ldots \odot_n x.l(\vec{v}) \ldots \odot_n \mathit{spawn}\, c_t(\vec{z}) \ldots \rangle$. The initial thread n_0, which is not instantiated from a thread class but is given directly (in case it starts in the component), has $\odot_{n_0}$ as augmentation. 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. We also omit the typing rules for the augmentation.

2.3.5 Use and change of contexts

Notation 1 *To facilitate the following definitions notationally, we make use of the following conventions. We abbreviate the triple of name contexts Δ, Σ, Θ as Φ, the context $\Delta, \Sigma, \Theta, E_\Delta, E_\Theta$ combining assumptions and commitments as Ξ, and write $\Xi \vdash C$ for $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$. We use syntactical variants analogously.*

The operational semantics is given by transitions between typed judgments

$$\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}.$$

The assumption context $\Delta, \Sigma; E_\Delta$ is an abstraction of the (not-present) environment, representing the potential behavior of all possible environments. The check whether the current assumptions are met in an incoming communication step is given in Definition 2. Note that for an incoming *call* label, $\mathit{fn}(a)$, the free names in a, includes the receiver o_r and the thread name.

Definition 2 (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*

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

Besides *checking* the connectivity assumptions 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 case of a spawn-label, also the name of 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' are object references transmitted by *scope extrusion,* Θ' the reference to the *lazily instantiated* objects, and Σ' contains new thread names. 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, whereas Θ' extends Θ, and Σ' extends the assumption and the commitment context. For outgoing communication, the situation is dual.

Definition 3 (Name context update: $\Phi + a$). *The* update $\acute{\Phi} = \Phi + a$ *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. $\acute{\Sigma} = \Sigma + \Sigma'$.

The notation $\odot_{\Sigma'}$ abbreviates $\odot_n$ for all thread names of Σ'. The update for outgoing communication is defined dually ($\odot_n$ of a spawn label is added to Δ instead of Θ, and the $\odot_{\Sigma'}$ resp. $\odot_{\Sigma' \setminus n}$ are added to Θ, instead of Δ).

Now the update of *connectivity*, concentrating again on incoming steps; the situation for outgoing communication is dual. Incoming communication may bring entities in connection which had been separate before, in particular it may merge object cliques. For the commitment context, this is formulated by adding the fact that the receiver of the communication now is acquainted with all transmitted arguments. See part (1) of Definition 4 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)). 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 clique as source, when the thread is new. In the operational rules, the update of Definition 4 is used where the sender is appropriately guessed under those circumstances.

Remains the treatment of *thread names* transmitted boundedly. Assume first 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 component 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 4 (Connectivity context update). *The update* $(\acute{E}_\Delta, \acute{E}_\Theta) = (E_\Delta, E_\Theta) + o_s \xrightarrow{a} o_r$ *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 given by:*

1. $\acute{E}_\Theta = E_\Theta + o_r \hookrightarrow fn(\lfloor a \rfloor)$.
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.*

Combining Definitions 3 and 4, we write $\acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r$ when updating the name and the connectivity at the same time.

Besides the connectivity check of Definition 2, we check the *static* assumptions, i.e., whether the transmitted values are of the correct types.

Definition 5 (Well-formedness and well-typedness of a label). *A label* $a = \nu(\Phi').\lfloor a \rfloor$ *is* well-formed, *written* $\vdash a$, *if* $dom(\Phi') \subseteq fn(\lfloor a \rfloor)$ *and if* Φ' *is a well-formed name-context for object names, i.e., no name bound in* Φ' *occurs twice.* Well-typedness *of a well-formed incoming core label* a *relative to the contexts* $\acute{\Phi} = \acute{\Delta}, \acute{\Sigma}, \acute{\Theta}$ *and with sender* o_s *and receiver* o_r *is given by the rules of Table 9. We use* $\acute{\Phi} \vdash o_s \xrightarrow{a} o_r : ok$ *as notation to assert well-typedness (where we assume that this additionally asserts well-formedness). For outgoing labels, the definition is dual.*

As for the update, we combine the checks for well-typedness and connectivity (Definition 2 and 5) into one assertion, written $\Xi \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : T$, resp. $\Xi \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : ok$, if the type does not matter.

2.3.6 Operational rules The operational rules for the external behavior are given in Table 10. Three CallI-rules deal with three different situations for incoming calls: A call reentrant on the level of the component, a call of a thread whose name is already known by the component, and a call of a thread new to the component. 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 type-checks and that the step is possible according to the (updated) connectivity assumptions. Remember

$$\frac{;\acute{\Sigma} \vdash n : \mathit{thread} \quad ;\acute{\Theta} \vdash o_r{:}c_r \quad ;\acute{\Delta}, \acute{\Theta} \vdash c_r : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad ;\acute{\Phi} \vdash \vec{v} : \vec{T} \quad a = n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?}{\acute{\Phi} \vdash \_ \xrightarrow{a} o_r : T}\ \text{LT-CallI}$$

$$\frac{\acute{\Theta} \vdash c_t : \vec{T} \to \mathit{thread} \quad \acute{\Sigma} \vdash n : \mathit{thread} \quad \acute{\Phi} \vdash \vec{v} : \vec{T} \quad a = \langle \mathit{spawn}\ n\ \mathit{of}\ c_t(\vec{v})\rangle?}{\acute{\Phi} \vdash \_ \xrightarrow{a} \_ : \mathit{thread}}\ \text{LT-SpawnI}$$

$$\frac{;\acute{\Delta} \vdash o_s : c_s \quad ;\acute{\Delta}, \acute{\Theta} \vdash c_s : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad ;\acute{\Phi} \vdash v : T \quad a = n\langle \mathit{return}(v)\rangle?}{\acute{\Phi} \vdash o_s \xrightarrow{a} \_ : T}\ \text{LT-RetI}$$

Table 9. Checking static assumptions

that the update from Ξ to $\acute{\Xi}$ includes guessing of connectivity, i.e., an element of non-determinism, when the sender of the call is unknown to the component. To deal with component entities (threads and objects) being created during the call, $C(\Theta', \Sigma')$ stands for $C(\Theta') \parallel C(\Sigma')$, where $C(\Theta')$ are the lazily instantiated objects mentioned in Θ'. Furthermore, for each thread name n' in Σ', a new component $n'\langle \mathit{stop}\rangle$ is included, written as $C(\Sigma')$.

For reentrant method calls in rule CALLI_1, the thread is blocked, i.e., it has left the component previously via an outgoing call. The object o_s that had been the target of the call is remembered as part of the augmented block syntax, and is used now to represent the sender's clique for 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 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.

In CALLI_2, the thread is not in the component, but the thread's name is already known. As a consequence, the component contains the entity $n\langle \mathit{stop}\rangle$. Unlike in CALLI_1, the program code contains no indication as to the origin of the call. Since the thread name 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 $n\langle \mathit{stop}\rangle$ for the code of a (deadlocked) incoming call. If $\Delta \vdash \odot_n$ and $n\langle \mathit{stop}\rangle$ 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 case $\textsc{CalleI}_{0,3}$ covers the situation, that a new thread n enters the component for the first time, as assured by the premise $\Sigma' \vdash n$. As in $\textsc{CalleI}_2$, we have no 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 by some environment clique, otherwise the cross-border instantiation would have been observed and the thread name would not be fresh. 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 SpawnI, we need again to know the origin of the call, i.e., the spawning clique. The situation is similar to the one for $\textsc{CalleI}_3$, in that the origin of the communication needs to be guessed. In $\textsc{CalleI}_3$, we use $\odot_n$ covering the situation where no actual calling object may be the source. Different from the situation of unknown caller is that here we obviously can 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 have 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. Definition 3 and 4) to cater for cross-border instantiation of the new thread. An incoming spawn action without known external objects is possible only in the very first step. It is covered by $\textsc{SpawnI}_0$ from Table 11.

Outgoing calls are dealt with in rule CalleO. 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. type checking: Starting 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 contain the threads instantiated from component thread classes and whose life starts at the component

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle?\quad t_{blocked} = let\ x{:}T' = o\ blocks\ for\ o_s\ in\ t \qquad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r\quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrowtail} o_r : T}{\Xi \vdash \nu(\Phi_1).(C \parallel n\langle t_{blocked}\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Phi_1).(C \parallel C(\Theta', \Sigma') \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ to\ o_s\ x; t_{blocked}\rangle)}\ \textsc{CalliI}_1$$

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

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle?\quad \Delta \vdash o\quad \Sigma' \vdash n\quad \acute{\Xi} = \Xi + o \xrightarrow{a} o_r\quad \acute{\Xi} \vdash \odot_n \overset{\lfloor a \rfloor}{\rightarrowtail} o_r : T}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle let\ x{:}T = o_r.l(\vec{v})\ in\ o_r\ returns\ to\ \odot_n\ x; stop\rangle}\ \textsc{CalliI}_3$$

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle!\quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi_1\quad \acute{\Phi}_1 = \Phi_1 \setminus \Phi'\quad \acute{\Delta} \vdash o_r\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\begin{array}{c}\Xi \vdash \nu(\Phi_1).(C \parallel n\langle let\ x{:}T = o_s\ o_r.l(\vec{v})\ in\ t\rangle) \xrightarrow{a} \\ \acute{\Xi} \vdash \nu(\acute{\Phi}_1).(C \parallel n\langle let\ x{:}T = o_s\ blocks\ for\ o_r\ in\ t\rangle)\end{array}}\ \textsc{CallO}$$

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

$$\frac{\begin{array}{c}a = \nu(n'{:}thread, \Phi').\langle spawn\ n'\ of\ c_t(\vec{v})\rangle!\quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi_1\quad \acute{\Phi}_1 = \Phi_1 \setminus \Phi' \\ \Delta \vdash c_t\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_{n'}\end{array}}{\Xi \vdash \nu(\Phi_1).(C \parallel n\langle let\ x{:}T = o_s\ spawn\ c_t(\vec{v})\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Phi}_1).(C \parallel n\langle let\ x{:}T = n'\ in\ t\rangle)}\ \textsc{SpawnO}$$

$$\frac{a = \nu(\Phi').\ n\langle return(v)\rangle?\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r\quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrowtail} o_r : T}{\Xi \vdash \nu(\Phi_1).(C \parallel n\langle let\ x{:}T = o_r\ blocks\ for\ o_s\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Phi_1).(C \parallel n\langle t[v/x]\rangle)}\ \textsc{RetI}$$

$$\frac{a = \nu(\Phi').\ n\langle return(v)\rangle!\quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi_1\quad \acute{\Phi}_1 = \Phi_1 \setminus \Phi'\quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\Xi \vdash \nu(\Phi_1).(C \parallel n\langle let\ x{:}T = o_s\ returns\ to\ o_r\ v\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Phi}_1).(C \parallel n\langle t\rangle)}\ \textsc{RetO}$$

$$\frac{\Delta \vdash c}{\Xi \vdash n\langle let\ x{:}c = new\ c\ in\ t\rangle \rightsquigarrow \Xi \vdash \nu(o{:}c).n\langle let\ x{:}c = o\ in\ t\rangle}\ \textsc{NewO}_{lazy}$$

Table 10. External steps

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. 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 remaining rules deal with returns 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, we denote the binding part of the label by $\nu(\Phi')$ 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 10, 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, and indeed the active thread as the subject of the interaction need not be the initial thread.

3 Legal traces

Next we present an independent characterization of the possible interface behavior. "Half" of the work has been done already by the careful design of the open semantics of Section 2.3.6, where the absent environment is represented abstractly by the name and connectivity contexts. For the legal traces, we analogously abstract away from the program code of the component, making the system completely symmetric. Remember that the assumption and commitment contexts in the operational semantics were used asymmetrically insofar, as the commitments were updated as over-approximation of the actual component, but

$$\frac{\begin{array}{c}\Xi_0 \vdash \mathit{static} \quad \Delta_0 \vdash \odot \quad \acute{\Xi} = \Xi_0 + \odot_n \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \overset{\lfloor a \rfloor}{\rightarrow} o_r : T \\ a = \nu(\Phi').\ n\langle \mathit{call}\ o_r.l(\vec{v})\rangle? \quad \Sigma' \vdash n\end{array}}{\begin{array}{l}\Xi_0 \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle \mathit{let}\, x{:}T = o_r.l(\vec{v})\ \mathit{in}\ o_r\ \mathit{returns\ to}\ \odot\ x; \mathit{stop}\rangle\end{array}}\ \textsc{CalliI}_0$$

$$\frac{\begin{array}{c}\Xi_0 \vdash \mathit{static} \quad \Theta_0 \vdash \odot \quad a = \nu(\Phi').\ n\langle \mathit{call}\ o_r.l(\vec{v})\rangle! \\ \Phi' = \mathit{fn}(\lfloor a \rfloor) \cap \Phi_1 \quad \acute{\Phi}_1 = \Phi_1 \setminus \Phi' \quad \acute{\Delta} \vdash o_r \quad \acute{\Xi} = \Xi_0 + o_s \xrightarrow{a} o_r\end{array}}{\begin{array}{l}\Xi_0 \vdash \nu(\Phi_1).(C \parallel n\langle \mathit{let}\, x{:}T = o_s\ o_r.l(\vec{v})\ \mathit{in}\ t\rangle) \xrightarrow{a} \\ \acute{\Xi} \vdash \nu(\acute{\Phi}_1).(C \parallel n\langle \mathit{let}\, x{:}T = o_s\ \mathit{blocks\ for}\ o_r\ \mathit{in}\ t\rangle)\end{array}}\ \textsc{CalliO}_0$$

$$\frac{\begin{array}{c}\Xi_0 \vdash \mathit{static} \quad \Delta_0 \vdash \odot \quad \acute{\Xi} = \Xi_0 + \odot \xrightarrow{a} \odot_n \quad \acute{\Xi} \vdash \odot \overset{\lfloor a \rfloor}{\rightarrow} \odot_n : \mathit{thread} \\ a = \nu(\Phi').\langle \mathit{spawn}\ n\ \mathit{of}\ c_t(\vec{v})\rangle? \quad \Theta_0 \vdash c_t \quad \Sigma' \vdash n\end{array}}{\Xi_0 \vdash C :\xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle c_t(\vec{v})\rangle}\ \textsc{SpawnI}_0$$

$$\frac{\begin{array}{c}\Xi_0 \vdash \mathit{static} \quad \Theta_0 \vdash \odot \quad a = \nu(n' : \mathit{thread}, \Phi').\langle \mathit{spawn}\ n'\ \mathit{of}\ c_t(\vec{v})\rangle! \\ \Phi' = \mathit{fn}(\lfloor a \rfloor) \cap \Phi_1 \quad \acute{\Phi}_1 = \Phi_1 \setminus \Phi' \quad \Delta_0 \vdash c_t \quad \acute{\Xi} = \Xi_0 + o_s \xrightarrow{a} \odot_{n'}\end{array}}{\begin{array}{l}\Xi_0 \vdash \nu(\Phi_1).(C \parallel n\langle \mathit{let}\, x{:}T = o_s\ \ \mathit{spawn}\, c_t(\vec{v})\ \mathit{in}\, t\rangle) \xrightarrow{a} \\ \acute{\Xi} \vdash \nu(\acute{\Phi}_1).(C \parallel n\langle \mathit{let}\, x{:}T = n'\ \mathit{in}\, t\rangle)\end{array}}\ \textsc{SpawnO}_0$$

Table 11. Initial external steps

not used in *checking* whether a component step, i.e., an outgoing communication, is possible as next interaction.

3.1 Linear representation

As mentioned, we characterize which traces can occur at all, and in a class-based setting it is crucial to take the connectivity into account. Besides that, the calls and returns of the thread must be "parenthetic", i.e., each return must have a matching call prior in the trace and we must take into account whether the thread is resident inside the component or outside.

Much of the work has been done already in the definition of the external step in the Tables 10-11: For incoming communication, for which the environment is responsible, we took care that only those steps are possible which may come from a realizable environment. As far as the reaction of the component was concerned, the code of the program is given; so the *reaction* of the component is not only realizable, but a fortiori "realized". To characterize when a given trace is *legal,* we need to require that the behavior of the component side, i.e., the outgoing communication, adheres to the dual discipline we imposed on the environment in the semantics.

The legal traces are specified by a system for judgments of the form

$$\Xi \vdash r \rhd s : \mathit{trace} \tag{5}$$

$$\frac{}{\vdash \epsilon : B_n^+}\ \text{B-Empty}^+ \qquad \frac{}{\vdash \epsilon : B_n^-}\ \text{B-Empty}^-$$

$$\frac{\vdash s_1 : B_n^+ \quad \vdash s_2 : B_n^+ \quad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : B_n^+}\ \text{B-II} \qquad \frac{\vdash s_1 : B_n^- \quad \vdash s_2 : B_n^- \quad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : B_n^-}\ \text{B-OO}$$

$$\frac{\vdash s : B_n^+}{\vdash \nu(\Phi).n\langle call\ o_2.l(\vec{v})\rangle?\ s\ \nu(\Phi').n\langle return(v)\rangle! : B_n^-}\ \text{B-IO}$$

$$\frac{\vdash s : B_n^-}{\vdash \nu(\Phi).n\langle call\ o_2.l(\vec{v})\rangle!\ s\ \nu(\Phi').n\langle return(v)\rangle? : B_n^+}\ \text{B-OI}$$

$$\frac{\vdash s' \downarrow_n : B_n^+ \qquad s \in \{s',\ \langle spawn\ n\ of\ c_t(\vec{v})\rangle?\, s'\}}{\vdash s : balanced_n^+}\ \text{B-Lift}^+$$

$$\frac{\vdash s' \downarrow_n : B_n^- \qquad s \in \{s',\ \langle spawn\ n\ of\ c_t(\vec{v})\rangle!\, s'\}}{\vdash s : balanced_n^-}\ \text{B-Lift}^-$$

Table 12. Balance

stipulating that under the type and relational assumptions Δ, Σ, and E_Δ and with the commitments Θ, Σ, and E_Θ, the trace s with history r is legal.

Roughly, the assertions used in the operational semantics are grouped into those for static typing and those for connectedness. Here, without the code of the program, we need an auxiliary assertion concerning the balance of calls and returns ("enabledness"). In the operational semantics, such an assertion was not even needed for the behavior of the environment, since, for instance, an incoming return step of a thread is possible only when the thread is blocked. Thus the program syntax takes care that calls and returns happen only in a well-balanced manner. Without code, we need an independent characterization.

3.1.1 Balance conditions We start with auxiliary definitions concerning the parenthetic nature of calls and returns of a legal trace. It is easy to see that, starting from an initial configuration, the operational semantics from Section 2.3.2 assures strict alternation of incoming and outgoing communication and additionally that there is no return without a preceding matching call. Later, we will need this property of traces for the characterization of legal traces.

Definition 6 (Balance). *The balance of a thread n in a sequence s of labels is given by the rules of Table 12. We write $\vdash s : balanced_n$ if $\vdash s : balanced_n^+$ or $\vdash s : balanced_n^-$. We call a (not necessarily proper) prefix of a balanced trace* weakly balanced. *We write $\vdash s : wbalanced_n^+$ if s is weakly balanced in n, i.e.,*

if $s \downarrow_n$ (the projection of s to thread n) is weakly balanced and its last label (if any) is an incoming communication; dually for $\vdash s : wbalanced_n^-$. The function pop_n for thread n on traces is defined as follows:

1. $pop_n\ \ s = \bot$, *if s is balanced in n.*
2. $pop_n(s_1 a s_2) = s_1 a$ *if* $a = \nu(\Phi).\ n\langle call\ o_r.l(\vec{v})\rangle?$ *and s_2 is* $balanced_n^+$.
3. $pop_n(s_1 a s_2) = s_1 a$ *if* $a = \nu(\Phi).\ n\langle call\ o_r.l(\vec{v})\rangle!$ *and s_2 is* $balanced_n^-$.

The source and the target of a communication step are needed for two reasons. First, to appropriately update the connectivity in a communication step. In the operational semantics we "augmented" the syntax of the call-, blocked-, and return-statements such that the information was readily available. Secondly, when defining the projection of a given trace to a clique of objects, we need to know which steps are interacting with the chosen clique and which not. In that case, we need the identity of the communication partners, without referring to the syntax of the program, but based on the sequence of interactions alone. This will also be necessary later when characterizing the legal traces.

Note that the communication labels alone do not contain enough information to determine their source and target. For call labels $\nu(\Phi).n\langle call\ o.l(\vec{v})\rangle$, only the target of the communication, the callee o, is mentioned, the caller remains anonymous. This is justified by the fact that the callee does not get hold of the identity of the caller. The identity of the caller can therefore not be observed and should thus not be mentioned in the interface behavior. Return labels $\nu(\Phi).n\langle return(\vec{v})\rangle$ do not mention any communication partner. However, the communication partners are determined by the communication history. For instance, the source of a return is target of the matching call. For a call it is assured that it leaves the same clique that the previous communication, call or return, has entered.

Based on a weakly balanced past, the following definition formalizes the notion of source and target of a communication event with the help of the function pop.[6]

Definition 7 (Sender and receiver). *Let $r\ a$ be the projection of a weakly balanced trace onto the thread n. Sender and receiver of a after history r are defined by mutual recursion and pattern matching over the following cases:*

$$
\begin{aligned}
sender(\nu(\Phi).n\langle call\ o_r.l(\vec{v})\rangle!) &= \odot_n \\
sender(\nu(\Phi).\langle spawn\ n\ of\ c_t(\vec{v})\rangle!) &= \bot \\
sender(r'\ a'\ \nu(\Phi).n\langle call\ o_r.l(\vec{v})\rangle!) &= receiver(r'\ a') \\
sender(r'\ a'\ \nu(\Phi).n\langle return(l(\vec{v}))\rangle!) &= receiver(pop_n(r'\ a'))
\end{aligned}
$$

$$
\begin{aligned}
receiver(\nu(\Phi).\langle spawn\ n\ of\ c_t(\vec{v})\rangle!) &= \odot_n \\
receiver(r\ \nu(\Phi).n\langle call\ o_r.l(\vec{v})\rangle!) &= o_r \\
receiver(r\ \nu(\Phi).n\langle return(\vec{v})\rangle!) &= sender(pop_n(r))
\end{aligned}
$$

For a being an incoming label, the definition is dual.

[6] Since we apply the definition onto the projection of a trace onto thread n, we omit in the function the thread name as parameter.

Note that source and target are well-defined. In particular, the recursive definition terminates. Furthermore, the weak balance of the argument assures that the call of *pop* yields a well-defined result and guarantees that the case distinction is exhaustive.

The premise $\Xi \vdash r \rhd a$ asserts that after r, the action a is enabled. Input enabledness checks whether, given a communication history, an incoming call is possible in the next step; analogously for output enabledness. To be input enabled, one checks against the last matching communication. If there is no such label, enabledness depends on where the thread started:

Definition 8 (Enabledness). *For a method call $\gamma = \nu(\Phi').n\langle call\ o_r.l(\vec{v})\rangle$, call-*enabledness *of γ after the history r and in the context Φ is defined as:*

$$\Phi \vdash r \rhd \gamma?\ \textit{if}\ \begin{array}{ll} pop_n & r = \bot \quad \textit{and} \quad \Delta \vdash \odot_n\ \textit{or} \\ pop_n & r = r'\gamma'! \end{array} \tag{6}$$

$$\Phi \vdash r \rhd \gamma!\ \textit{if}\ \begin{array}{ll} pop_n & r = \bot \quad \textit{and} \quad \Theta \vdash \odot_n\ \textit{or} \\ pop_n & r = r'\gamma'? \end{array} \tag{7}$$

For labels $\gamma = \nu(\Phi').n\langle return(v)\rangle$, the assertion $\Phi \vdash r \rhd \gamma!$ abbreviates $pop_n\, r = r'\nu(\Phi'').n\langle call\ o_2.l(\vec{v})\rangle?$, and dually for incoming returns $\gamma?$. Spawn labels are always enabled.

We also say, the thread is *input-call enabled* after r if $\Phi \vdash r \rhd \gamma?$ for some incoming call label, respectively *input-return* enabled in case of an incoming return label. The definitions are used dually for output-call enabledness and output-return enabledness. When leaving the kind of communication unspecified we just speak of input-enabledness or output-enabledness. Note that return-enabledness implies call-enabledness, but not vice versa.

We further combine enabledness and determining sender and receiver (cf. Definitions 7 and 8) into the following notation:

$$\Phi \vdash r \rhd o_s \xrightarrow{a} o_r\ . \tag{8}$$

3.1.2 Legal traces system Table 13 specifies *legality* of traces, combining all mentioned conditions, basically *type checking, connectivity,* and *balance.* We use the same conventions and notations as for the operational semantics (cf. Notation 1).

As base case, the empty future is always legal, and distinguishing according to the nature of the first action a of the trace, the rules check whether a is possible, i.e., whether it is enabled, well-typed and adheres to the restrictions imposed by the connectivity contexts. Furthermore, the contexts are updated appropriately and the rules recur checking the tail of the trace. The rules are symmetric wrt. incoming and outgoing communication.

The L-CalleI-rules works similar to the three CalleI-rules in the semantics. A difference is that sender and receiver are now not taken from the code, but determined from the past interaction. The premise $\Xi \vdash r \rhd o_s \xrightarrow{a} o_r$ checks

$$\Xi \vdash r \rhd \epsilon : \mathit{trace} \qquad \text{L-Empty}$$

$$\frac{\Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} o_r : T \quad a = \nu(\Phi').\ n\langle \mathit{call}\ o_r.l(\vec{v})\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : \mathit{trace}}{\Xi \vdash r \rhd a\ s : \mathit{trace}}\ \text{L-CalleI}_{1,2}$$

$$\frac{\Sigma' \vdash n \quad \Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \Delta \vdash o \quad \acute{\Xi} = \Xi + o \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} o_r : T \quad a = \nu(\Phi').\ n\langle \mathit{call}\ o_r.l(\vec{v})\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : \mathit{trace}}{\Xi \vdash r \rhd a\ s : \mathit{trace}}\ \text{L-CalleI}_{0,3}$$

$$\frac{\acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_n \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} \odot_n \quad \Sigma' \vdash n \quad \Delta \vdash o_s \quad a = \nu(\Phi').\langle \mathit{spawn}\ n\ \mathit{of}\ c_t(\vec{v})\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : \mathit{trace}}{\Xi \vdash r \rhd a\ s : \mathit{trace}}\ \text{L-SpawnI}$$

$$\frac{\Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \overset{\lfloor a \rfloor}{\rightarrow} o_r : T \quad a = \nu(\Phi').\ n\langle \mathit{return}(v)\rangle? \quad \acute{\Xi} \vdash r\ a \rhd s : \mathit{trace}}{\Xi \vdash r \rhd a\ s : \mathit{trace}}\ \text{L-RetI}$$

Table 13. Legal traces (dual rules omitted)

whether the incoming call a is enabled and determines sender and receiver. The receiver o_r, of course, is mentioned directly, but o_s is calculated from the history r. Note especially that if the call is the *first* activity of a thread, i.e., the name of the thread is transmitted boundedly, o_s equals $\odot_n$, which corresponds to the situation of CalleI$_3$ resp. CalleI$_0$. Similarly, when thread n is balanced in r, the sender is determined as $\odot_n$, as well, and the treatment corresponds to CalleI$_2$. For incoming spawn labels (cf. rule L-SpawnI), connectivity requires that there *exists* an environment clique (possibly a $\odot_{n'}$) as spawner. This is directly required in the premise $\Delta \vdash o_s$. Note also that there is no premise requiring enabledness; for a new thread, it makes no sense to require "balance" nor to determine the spawner by consulting the past.

Remark 3 (Anonymous spawner and exchange of thread identities). For outgoing thread creation, the spawning clique of objects is non-deterministically guessed by rule L-SpawnO (for incoming communication, the situation is dual). The identity of the originator, however, needs not be remembered. Cross-border thread creation is only one instance where the identity of a new thread is communicated from component to environment or vice versa. Alternatively, the thread name may be communicated by ordinary scope extrusion. Also in this case neither the spawning thread nor the spawning clique of objects is known, and the latter has to be guessed. From the perspective of the component, we distinguish the situation where the thread name enters the component as bearer of the activity, or simply in argument or object position.

In subject position, a thread can enter the component in two ways: by cross-border thread creation, as mentioned, and by an incoming call. Obviously, an active thread cannot enter the component for the first time via a return. In case of an incoming call introducing a new thread, the clique of the originator is guessed via rule L-CALLI$_3$. In that situation the sender is $\odot_n$, where $\odot_n$ is a new symbol. This represents the non-deterministic guessing and the provisos of the rule check whether the call is possible by checking the connectivity of $\odot_n$.

When a thread name extrudes its scope via ordinary message passing, i.e., in object position of a communication, its identity is treated as an ordinary name in the connectivity contexts, except that $o_1 \hookrightarrow n$ and $o_2 \hookrightarrow n$ does not imply $o_1 \leftrightharpoons o_2$. □

3.2 Branching system

The legal traces and also the operational semantics represent the worst-case assumptions about the object connectivity. Since we can pass thread names in argument position, also the connectivity wrt. those names has to be considered. However, more important is the connectivity of *objects*: only for object identities, the transitive and symmetric closure of the relation is considered. To put it differently: *cliques* are equivalence classes of objects, but not threads. The cliques of objects are not only relevant to characterize the legal traces, they indeed lie at the heart of the semantics: actions concerning two separate component cliques can occur in either order, and traces belonging to different environment cliques can be swapped without observable difference. Since new cliques can be created and especially existing cliques can merge by communication, it means that the semantics is *tree structured.* Note that the tree branches "into the past". In particular the branching structure of the semantics has nothing to do with branching due to non-determinism. The work [4] investigates the connectivity-based semantics in a deterministic, single-threaded setting.

Next we characterize the possible interface behavior where the *derivation* itself represents the *branching* nature of the semantics.

Unlike before, the connectivity of objects is not explicitly represented by connectivity contexts; instead, the *tree structure* of the derivation itself represents the connectivity and its change. There are two variants of the derivation system, one from the perspective of the *component,* and one from the perspective of the *environment.* Each derivation corresponds to a *forest,* with each root representing a component, resp. environment clique at the end. In judgments of the form

$$\Delta, \Sigma \vdash_\Theta r \rhd s : \mathit{trace}\, \Theta, \Sigma, \tag{9}$$

r represents the history, and s the future interaction. We write $\vdash_\Theta$ to indicate that legality is checked from the perspective of the component. From that perspective, we maintain as invariant that the context Θ represents one single clique. Thus the connectivity among objects of Θ needs no longer be remembered. What needs to be remembered still are the thread names known by Θ and the cross-border connectivity, i.e., the acquaintance of the clique represented

by Θ with objects of the environment. This is kept in Δ resp. Σ. Note that this corresponds to the environmental objects mentioned in $E_\Theta \subseteq \Theta \times (\Theta + \Delta + \Sigma)$, *projected* onto the component clique under consideration, in the linear system. The connectivity of the environment is *ignored* which means that the system of Table 14 *cannot* assure that the environment behaves according to a possible connectivity. On the other hand, dualizing the rules checks whether the environment adheres to possible connectivity.

Definition 9 (Join of contexts). *Given a pair of contexts $\Phi_1 = \Delta_1, \Sigma_1, \Theta_1$ and $\Phi_2 = \Delta_2, \Sigma_2, \Theta_2$. We define the* join *of Φ_1 and Φ_2 from the perspective of the component as follows:*

$$\Phi = \Phi_1 \oplus_\Theta \Phi_2 \quad \textit{iff} \quad \Phi = \Delta, \Sigma, \Theta \quad \textit{where} \quad \begin{array}{l} \Delta = \Delta_1 \cup \Delta_2 \\ \Sigma = \Sigma_1 \cup \Sigma_2 \\ \Theta = \Theta_1 + \Theta_2 \ . \end{array} \tag{10}$$

The index Θ of $\oplus_\Theta$ is not meant as argument, but indicates that the sum is interpreted from the perspective of the component. From that perspective, the commitment contexts Θ_1 and Θ_2 are merged *disjointly,* while the thread names from Σ_1 and Σ_2, and the references from the environment, i.e., from Δ_1 and Δ_2 not necessarily so, but by ordinary union.[7] Note that $\oplus_\Theta$ is a *partial* operation; it is undefined if the join $\Theta_1 + \Theta_2$ or $\Delta_1 \cup \Delta_2$ or $\Sigma_1 \cup \Sigma_2$ fails. The operation is symmetric and associative. We abbreviate the join of a finite number of contexts by $\bigoplus_\Theta \Phi_i$. The empty sum corresponds to the empty context.

Now to Table 14. Rule L-CallI deals with incoming calls. The call is possible only when the thread is input call enabled after the current history. Incoming communication may *update* the component connectivity, in that new cliques may be created or existing cliques may merge. The merging of component cliques is now represented by a branching of the proof system. Leaves of the resulting forest correspond to freshly created cliques. In L-CallI, the context Θ in the premise corresponds to the merged clique, the Θ_i below the line to the still split cliques before the merge. The Θ_i's form a partitioning of the component objects before communication, Θ is the disjoint combination of the Θ_i's plus the lazily instantiated objects from Θ'. For the cross-border connectivity, the different component cliques Θ_i may of course share acquaintance; thus, the parts Δ_i and Σ_i are not merged disjointly, but by ordinary "set" union. These restrictions are covered by the definition of the (partial) operation $\bigoplus \Phi_i$. In the premise $a_j = {\downarrow_{\Theta_j}}$, the $a \downarrow_{\Theta_j}$ denotes the projection of the label a onto the component clique Θ_j. This means, a_j interprets the *new* names, i.e., the binding part of the label, locally from the perspective of Φ_j. The condition $\Theta_j \vdash \lfloor a \rfloor$ requires that there *exists* a name in $\lfloor a \rfloor$ contained in Θ_j. The negative assertion $\Theta \nvdash \lfloor a \rfloor$ (in L-SkipI) is meant as: *no* name from $\lfloor a \rfloor$ is contained in Θ.

[7] Technically, of course, the contexts are not sets but syntactical entities of the calculus; however, the invariants enforced by the type system and maintained by the semantics allows to consider them as finite mappings from names to types.

$$\frac{\Phi = \bigoplus_{\Theta} \Phi_j \quad \Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Phi} = \Phi_0, \Phi + a \quad \acute{\Phi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r :ok \quad \forall j.\ a_j = a \downarrow_{\Theta_j} \wedge \Theta_j \vdash \lfloor a \rfloor \quad a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle ? \quad r \neq \epsilon \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi_1 \vdash r \rhd a_1\ s : trace \quad \ldots \quad \Phi_k \vdash r \rhd a_k\ s : trace}\ \text{L-CALLI}$$

$$\frac{\Phi_0 \vdash \epsilon \rhd \odot_n \xrightarrow{a} o_r \quad \acute{\Phi} = \Phi_0 + a \quad \acute{\Phi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r :ok \quad a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle ? \quad r = \epsilon \quad \Delta_0 \vdash \odot \quad \acute{\Phi} \vdash a \rhd s : trace}{\Phi_0 \vdash \epsilon \rhd a\ s : trace}\ \text{L-CALLI}_0$$

$$\frac{\Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \Theta \vdash o_s \quad \acute{\Phi} = \Phi + a \quad \acute{\Phi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r :ok \quad a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle ! \quad r \neq \epsilon \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi \vdash r \rhd a\ s : trace}\ \text{L-CALLO}_{1,2}$$

$$\frac{\Phi \vdash r \rhd \odot_n \xrightarrow{a} o_r \quad \Theta \vdash o \quad \acute{\Phi} = \Phi + a \quad \acute{\Phi} \vdash o \xrightarrow{\lfloor a \rfloor} o_r :ok \quad \Sigma' \vdash n \quad a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle ! \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi \vdash r \rhd a\ s : trace}\ \text{L-CALLO}_{0,3}$$

$$\frac{\Phi = \bigoplus_{\Theta} \Phi_j \quad \Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Phi} = \Phi + a \quad \acute{\Phi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r :ok \quad \forall j.\ a_j = a \downarrow_{\Theta_j} \wedge \Theta_j \vdash \lfloor a \rfloor, o_r \quad a = \nu(\Phi').\ n\langle return(v)\rangle ? \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi_1 \vdash r \rhd a_1\ s : trace \quad \ldots \quad \Phi_k \vdash r \rhd a_k\ s : trace}\ \text{L-RETI}$$

$$\frac{\Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Phi} = \Phi + a \quad \acute{\Phi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r :ok \quad a = \nu(\Phi').\ n\langle return(v)\rangle ! \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi \vdash r \rhd a\ s : trace}\ \text{L-RETO}$$

$$\frac{\Phi = \bigoplus_{\Theta} \Phi_j \quad \acute{\Phi} = \Phi_0, \Phi + a \quad \acute{\Phi} \vdash o \xrightarrow{\lfloor a \rfloor} \odot_n :ok \quad \Sigma' \vdash n \quad \Delta \vdash o \quad \forall j.\ a_j = a \downarrow_{\Theta_j} \wedge \Theta_j \vdash \lfloor a \rfloor \quad a = \nu(\Phi').\langle spawn\ n\ of\ c_t(\vec{v})\rangle ? \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi_1 \vdash r \rhd a_1\ s : trace \quad \ldots \quad \Phi_k \vdash r \rhd a_k\ s : trace}\ \text{L-SPAWNI}$$

$$\frac{\acute{\Phi} = \Phi + a \quad \acute{\Phi} \vdash o \xrightarrow{\lfloor a \rfloor} \odot_n :ok \quad \Sigma' \vdash n \quad \Theta \vdash o \quad a = \nu(\Phi').\langle spawn\ n\ of\ c_t(\vec{v})\rangle ! \quad \acute{\Phi} \vdash r\ a \rhd s : trace}{\Phi \vdash r \rhd a\ s : trace}\ \text{L-SPAWNO}$$

$$\frac{a = \gamma ? \quad \Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \Theta \not\vdash \lfloor a \rfloor, o_r \quad \Phi \vdash ra \rhd s : trace \quad r \neq \epsilon}{\Phi \vdash r \rhd s : trace}\ \text{L-SKIPI}$$

$$\frac{a = \gamma ! \quad \Phi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \Theta \not\vdash o_s \quad \Phi \vdash ra \rhd s : trace \quad r \neq \epsilon}{\Phi \vdash r \rhd s : trace}\ \text{L-SKIPO}$$

$$\Phi \vdash r \rhd \epsilon : trace \qquad \text{L-EMPTY}$$

Table 14. Legal traces, branching on Θ

For *outgoing* communication, no branching occurs. E.g. in the L-CallO-rules, the contexts Θ, Σ and Δ, Σ are updated appropriately with the object references transmitted boundedly. From the component's perspective, the contexts are needed to check whether the call is possible, especially concerning connectivity, for instance whether the caller knows the callee and all of the arguments. In the linear system, this was done by consulting E_Θ, as specified in Definition 2. These checks are now covered by the *static typing* premises, which assure that the receiver o_r is contained in $\acute{\Delta}$, and the arguments $\vec{v}$ in $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}(= \acute{\Phi})$. Otherwise, $\acute{\Phi} \vdash \lfloor a \rfloor : ok$ would fail. As an aside: In L-Call$_{0,3}$, the argument o is not needed for the type check $\acute{\Phi} \vdash o \xrightarrow{\lfloor a \rfloor} o_r$.

The return rules work similarly. They are formulated again a bit more general than necessary to stress the similarity between calls and returns as far as the information flow is concerned. Note in particular the condition $\Theta_j \vdash \lfloor a \rfloor, o_r$ in L-RetI: the receiver mentioned in addition to the references occurring in $\lfloor a \rfloor$ to merge also the clique of o_r, even if it is not mentioned in the label. The initial rules start[8] from an empty history and in static contexts, i.e., contexts without mentioning objects. The derivation ends at the root of a tree, when the future trace is empty (cf. rule L-Empty).

The skip-rules allow actions a not belonging to the component clique under consideration to be omitted from the component's "future" (interpreting the rule from bottom to top). The distinction is made according to the sender resp. the receiver of the communication (cf. rule L-SkipO resp. L-SkipI).It condition $r \neq \epsilon$ assures that the derivation cannot start with skip rules; this would make it possible to "skip over" a complete trace.

Definition 10 (Legal traces, tree system). *We write $\Phi_0 \vdash_\Theta t : trace$, if there exists a derivation forest using the rules of Table 14 with roots $\Delta_i, \Sigma_i \vdash t \rhd \epsilon : trace\, \Theta_i, \Sigma_i$ and a leaf justified by one of the initial rules. Using the dual rules, we write $\vdash_\Delta$ instead of $\vdash_\Theta$. We write $\Delta_0 \vdash_{\Delta \wedge \Theta} t : trace\, \Theta_0$, if there is a pair of derivations in the $\vdash_\Delta$- and the $\vdash_\Theta$- system with a consistent pair of root judgments (cf. Definition 11 below). We refer with f_Δ to the derivation forest for Δ, and dually for f_Θ.*

Remark 4 (Update of the assumptions). As in the linear setting, incoming communication does not only update the commitment context, here especially by merging previously separate Θ_i, but also the assumption contexts, namely wrt. freshly introduced references. Concerning the already known names, no new information is added. We can phrase this property later using the notion of conservative extension (cf. Definition 16). In the system from the perspective of the component, we *ignore* connectivity of the environment. Therefore we need not take care of that. □

[8] They "start" when interpreting the rules to work forward through the trace.

3.3 Equivalence of the legal traces representations

Next we show equivalence of the two representations of the interface behavior. To distinguish the rules for the different derivation systems, we write L-CALLI$^{\Delta}$ to denote the L-CALLI-rule in the branching system from the perspective of the environment; dually by superscripting Θ for rules from the perspective of the component. Rules from the linear system are used without superscript.

Definition 11 (Consistent contexts and judgments). *Assume sets of triples of contexts $(\Delta_i, \Sigma_i, \Theta_i)$ with the usual convention that the Σ's contain the bindings for thread names and the Δ's and Θ's those for environment resp. component names other than thread names. We call the set of contexts $\{(\Delta_i, \Sigma_i, \Theta_i) \mid i \in I\}$ to be* Δ-consistent*, if*

1. *Δ_i, Θ_i, and Σ_i are pairwise disjoint.*
2. *$\Delta_i \cap \Delta_{i'} = \emptyset$ (where $i \neq i'$).*

Dually the notion of Θ-consistency for a set of contexts $\{(\Delta_j, \Sigma_j, \Theta_j) \mid j \in J\}$, where $\Theta_j \cap \Theta_{j'} = \emptyset$ is required instead. In the rest of the paper, we leave the index sets I over with i ranges, implicit, i.e., we omit stating $i \in I$. By convention, we use $i \in I$ when describing contexts from the perspective of Δ, and $j \in J$ from the perspective of Θ. In a similar spirit we allow ourselves to write shorter $\{\Phi_i\}$ for sets of contexts $\{\Phi_i \mid i \in I\}$, etc.

A pair of sets of contexts $(\{(\Delta_i, \Sigma_i, \Theta_i)\}, \{(\Delta_j, \Sigma_j, \Theta_j)\})$ is called consistent*, if $\{(\Delta_i, \Sigma_i, \Theta_i)\}$ resp. $\{(\Delta_j, \Sigma_j, \Theta_j)\}$ is Δ-consistent, resp. Θ-consistent, and additionally*

3. *$\sum \Delta_i = \bigcup \Delta_j$, $\sum \Theta_j = \bigcup \Theta_i$, and $\bigcup \Sigma_i = \bigcup \Sigma_j$.*

A pair of sets of judgments $\{\Delta_i, \Sigma_i \vdash_{\Delta} r_i \rhd s_i : \mathit{trace}\, \Theta_i, \Sigma_i\}$, $\{\Delta_j, \Sigma_j \vdash_{\Theta} r_j \rhd s_j : \Theta_j, \Sigma_j\}$ is called consistent, if the corresponding condition holds for the assumption/commitment contexts.

Clearly, there is a *one-to-one* correspondence between consistent sets of contexts $\{(\Delta_i, \Sigma_i, \Theta_i)\}$ and $\{(\Delta_j, \Sigma_j, \Theta_j)\}$ and an assumption/commitment context of the form $(\Delta, \Sigma; E_{\Delta}, \Theta, \Sigma; E_{\Theta})$ as used in the linear system. The contexts $(\Delta_i, \Sigma_i, \Theta_i)$ correspond to the projection of the connectivity to the cliques according to $\Delta, \Sigma; E_{\Delta}$, and dually the $(\Delta_j, \Sigma_j, \Theta_j)$ to the projection onto the cliques according to $\Theta, \Sigma; E_{\Theta}$. The next definition recovers the representation as used in the linear system from the context representation as used in the branching system, using the $\oplus$-operator from Definition 9.

Definition 12 (Branching $\rightarrow$ linear). *Given a Δ-consistent set $\{\Phi_i\}$ of name contexts, the corresponding "linear" context $\mathcal{L}_{\Delta}(\{\Phi_i\})$ is defined as Φ, E_{Δ} where $\Phi = \bigoplus_{\Delta} \Phi_i$ and $E_{\Delta} = \sum_i \Delta_i \times (\Delta_i + \Sigma_i + \Theta_i)$. The definition of $\mathcal{L}_{\Theta}(\{\Phi_j\})$ for a Θ-consistent set $\{\Phi_j\}$ is defined dually. Given a consistent pair of contexts $\{\Phi_i\}, \{\Phi_j\}$, then $\mathcal{L}(\{\Phi_i\}, \{\Phi_j\})$ is defined as $\Phi; E_{\Delta}; E_{\Theta}$ with $\Phi = \bigoplus_{\Delta} \Phi_i$ and the connectivity contexts E_{Δ} and E_{Θ} as above. Note that for the name contexts Φ, we have $\bigoplus_{\Delta} \Phi_i = \bigoplus_{\Theta} \Phi_j$.*

The next definition updates the contexts wrt. a label for all judgments in a $\vdash_{\Delta\wedge\Theta}$-derivation at a given level at once. It simply applies the contexts update from Definition 3 to all those contexts in a branching derivation.

Definition 13 (Member/connectivity update). *Let $\{\Phi_i\}$, resp. $\{\Phi_j\}$, be a Δ-consistent, resp. Θ-consistent, set of contexts.*

1. *(a) Given an incoming label a with sender o_s. Then $\{\Phi_i\} + o_s \xrightarrow{a}$ is defined as $\{\acute{\Phi}_i\}$, where $\acute{\Phi}_i = \Phi_i + a$ when $\Delta_i \vdash o_s$, and $\acute{\Phi}_{i'} = \Phi_{i'}$ otherwise, and where it is assumed that $\Delta_i \vdash o_s$ for one Δ_i.*
   (b) Dually for incoming label a with receiver o_r, then $\{\Phi_j\} + \xrightarrow{a} o_r$ is given as $\{\acute{\Phi}_j\}$, where $\acute{\Phi}_j = \Phi_j + a$ when $\Phi_j \not\vdash \lfloor a \rfloor, o_r$ and $\acute{\Phi}_{j'} = \Phi_{j'}$ otherwise.
   For outgoing labels, the definition applies dually.
2. *When $\{\Phi_i\}$ and $\{\Phi_j\}$ are additionally consistent, we write $(\{\Phi_i\}, \{\Phi_j\}) + o_s \xrightarrow{a} o_r$ for the combination.*

Note that the update considered in isolation may not preserve consistency. For instance in case (1a), the update from Φ_i to $\acute{\Phi}_i = \Phi_i + a$ for the contexts Φ_i with $\Delta_i \vdash o_s$ may violate the requirement that the $\acute{\Delta}_i$ is disjoint from all other $\acute{\Delta}_{i'}$. This happens if the communication originating from Δ_i carries names of environment objects which are already present in some other $\Delta_{i'}$. The update in the branching version of the legal trace systems, however, is used in combination with the type check conditions which assure that consistency is preserved (cf. Lemma 3).

The next two straightforward lemmas state that the respective definitions of context update coincide in the branching and in the linear system. Both are straightforwardly shown by simple calculations. The + on the (11) (resp. (12) is the update from Definition 13 on the equation's left-hand side, and from Definition 3 on the right.

Lemma 1 (Name update). *Given a Δ-consistent set of contexts $\{\Phi_i\}$ and a Θ-consistent set of contexts $\{\Phi_j\}$. Let furthermore a be an incoming communication with sender o_s and receiver o_r. Then*

$$\bigoplus\nolimits_{\Delta}(\{\Phi_i\} + o_s \xrightarrow{a}) = (\bigoplus\nolimits_{\Delta} \Phi_i) + a \tag{11}$$

and dually

$$\bigoplus\nolimits_{\Theta}(\{\Phi_j\} + \xrightarrow{a} o_r) = (\bigoplus\nolimits_{\Theta} \Phi_j) + a \ . \tag{12}$$

If furthermore $\{\Phi_i\}$ and $\{\Phi_j\}$ are consistent, the results of the two equations coincide. Diagrammatically:

$$\begin{array}{ccc} \acute{\Phi}_i, \acute{\Phi}_j & \xrightarrow{\oplus_\Delta} & \acute{\Phi} \\ {\scriptstyle o_s \xrightarrow{a} o_r}\uparrow & & \uparrow{\scriptstyle a} \\ \Phi_i, \Phi_j & \xrightarrow{\oplus_\Delta} & \Phi \end{array} .$$

Note that the update in the branching system (the up-arrow on the left) unlike the name context update in the linear system (on the left) needs the sender resp. the receiver of a (cf. Definition 13(1a) resp. (1b)) since the partitioning in $\{\Phi_i\}$ resp. $\{\Phi_j\}$ contains also connectivity information. For outgoing communication, the lemma holds dually.

Proof. By straightforward calculation. □

Lemma 2 (Connectivity update). *Let $\{\Phi_i\}, \{\Phi_j\}$ be a consistent pair of sets of contexts. Then*

$$\mathcal{L}(\{\Phi_i\}, \{\Phi_j\}) + o_s \xrightarrow{a} o_r = \mathcal{L}\left((\{\Phi_u\}, \{\Phi_j\}) + o_s \xrightarrow{a} o_r\right) \ . \tag{13}$$

Proof. Straightforward. □

The next lemma states that at each level of a derivation in the branching system, the contexts are consistent.

Lemma 3 (Consistency). *Assume $\Delta_0 \vdash_{\Delta\wedge\Theta} t : \mathit{trace}\,\Theta_0$. Then there exists a derivation, i.e., a pair of derivation forests f_Δ, f_Θ where all sets of judgments $f_\Delta(k), f_\Theta(k)$ are consistent, for all $0 \leq k \leq n$, where k is the length of trace t, and $f_\Delta(k)$ is the forest up-to height k, counting from the leaf justified by one of the rules for initial interaction.*

Proof. By straightforward induction on the length k of the pair of derivations (cf. Table 14). Initially, for $k = 0$ (at the bottom-most leaf), the context Φ_0 (either for f_Δ or f_Θ depending on whether $\Delta_0 \vdash \odot$ or $\Theta_0 \vdash \odot$) is consistent, as it contains no object or thread references except $\odot$.

For an induction step, we take an incoming call as example. For $\vdash_\Delta$, the relevant rules are the L-CALLI$^\Delta$-rules and L-SKIPI$^\Delta$, which are the duals of the L-CALLO$^\Theta$-rules and L-SKIPO$^\Theta$ from Table 14, i.e., for f_Δ, no branching occurs. We treat both rules for incoming calls at the same time, writing L-CALLI$^\Delta$ for them both (and the same later for L-CALLI$^\Theta$). Let Φ_i be the context changed by L-CALLI$^\Delta$, and $\Phi_{i'}$ those (if any) which are left unchanged by L-SKIPI$^\Delta$. Note that there can be at most one instance of L-CALLI$^\Delta$ at that step —all other trees must be handled by the skip rule— since by induction the contexts before the step are Δ-consistent. So Δ-consistency is potentially violated by the update $\acute{\Phi}_i = \Phi_i + a$, in particular, the update from Δ_i to $\acute{\Delta}_i$ by scope extrusion may violate the disjointness-requirement for $\acute{\Delta}_i$ and $\acute{\Delta}_{i'}$. The type-checking premise $\acute{\Phi}_i \vdash o \xrightarrow{\lfloor a \rfloor} o_r : ok$ assures that all names of environment objects mentioned in $\lfloor a \rfloor$ are either already covered in Δ_i (no scope extrusion), and hence by induction not in conflict with any $\acute{\Delta}_{i'}$, or are transmitted under a ν-binder (scope extrusion) and hence by renaming are different from those from any $\acute{\Delta}_{i'}$.

For $\vdash_\Theta$ and Θ-consistency, the relevant rules are the L-CALLI$^\Theta$-rules and L-SKIP$^\Theta$. Let Φ_j be the context updated by L-CALLI$^\Theta$ and $\Phi_{j'}$ the contexts (if any) left unchanged by the skip rule. Note that there can be at most one instance of L-CALLI$^\Theta$ since by induction, the contexts before the step are Θ-consistent. In particular, the corresponding Θ_k and $\Theta_{j'}$ contexts are disjoint,

and hence, the premise $\Theta_j \vdash \lfloor a \rfloor$ applies at most once. So the only update which potentially violates Θ-consistency is the one from the $\Phi_j^1 \dots \Phi_j^m$ in the branches to $\acute{\Phi}_j = \oplus\Phi_j^k + a$ in the premise of L-CALL$^\Theta$. In particular, the part $\acute{\Theta}_j$ from $\acute{\Phi}_j$ must be disjoint from all other $\acute{\Theta}_{j'} = \Theta_{j'}$. This is assured since *no* object reference from any $\Theta_{j'}$ is added by the "$+a$" to $\oplus\Phi_j^k$, which is enforced by the premises $\Theta'_j \nvdash \lfloor a \rfloor$ resp. $\Theta_j^k \vdash \lfloor a \rfloor$, which distinguishes between L-CALLI$^\Theta$ and L-SKIP$^\Theta$.

For consistency, the argument works similarly. □

Lemma 4 (No change). *Let* $(\{\Phi_i\}, \{\Phi_j\})$ *be consistent and* $(\{\acute{\Phi}_i\}, \{\acute{\Phi}_j\}) = (\{\Phi_i\}, \{\Phi_j\}) + o_s \xrightarrow{a} o_r$. *If* $\{\acute{\Phi}_i\} = \{\Phi_i\}$, *then also* $\{\acute{\Phi}_j\} = \{\Phi_j\}$ *(and vice versa).*

Proof. Straightforward. □

Lemma 5 (Exactly one sender). *Assume* $\Phi_0 \vdash_{\Delta\wedge\Theta} t : trace$, *with a pair of derivation forests* f_Δ, f_Θ. *Assume* $\Phi \vdash_\Delta r \rhd as : trace$, *where* a *is an incoming call. Then exactly one judgment* $\Phi \vdash_\Delta ra \rhd s : trace$ *of* f_Δ *is justified by one of the* L-CALLI$^\Delta$*-rules (all others by* L-SKIPI$^\Delta$*).*

Proof. First we show that *at least* one judgment is justified by L-CALLI$^\Delta$. Assume for a contradiction, that there is *no* instance of that rule, which means that for f_Δ, the steps in the forest are all justified by L-SKIPI$^\Delta$ (other rules do not apply because of the form of the label a). The fact that $\Delta_i \vdash o_s$ for some Δ_i (where sender o_s is as determined by $\Phi \vdash r \rhd o_s \xrightarrow{a} o_r$) yields the contradiction. The assumption that there is more than one instance of L-CALLI$^\Delta$ implies that $\acute{\Delta}_i \vdash o_s$ and $\acute{\Delta}_{i'} \vdash o_s$, which contradicts the fact that the set of contexts in f_Δ are Δ-consistent (Lemma 3). □

Lemma 6 (Branching = linear). *For each trace* t *we have:* $\Phi_0 \vdash_{\Delta\wedge\Theta} t : trace$ *iff.* $\Xi_0 \vdash_{\Delta,\Theta} t : trace$ *in the linear system (where* Ξ_0 *equals* Φ_0 *as there are no dynamic references yet). More succinctly*

$$\vdash_{\Delta\wedge\Theta} = \vdash_{\Delta,\Theta} , \tag{14}$$

where $\vdash_{\Delta,\Theta}$ *represents derivability in the linear system.*

Proof. There are two directions to show.

Case: $\Rightarrow$

We are given $\Phi_0 \vdash_\Delta t : trace$ and $\Phi_0 \vdash_\Theta t : trace$, i.e., two derivation forests; let us call them f_Δ and f_Θ. The first one from the perspective of the environment with $\Delta_i, \Sigma_i \vdash_\Delta t \rhd \epsilon : trace\, \Theta_i, \Sigma_i$ as roots of the derivation trees. Likewise the forest wrt. the component.

Given the two forests, we construct inductively the corresponding linear derivation for $\Phi_0 \vdash_{\Delta,\Theta} t : trace$, by considering both forests at the same time. The construction proceeds *"from bottom to top"*, i.e., it begins with the leaves of f_Δ and f_Θ in the initial, static contexts. Let's further denote by $f_\Delta(n-k)$ the sub-forest of f_Δ with distance k from the the roots, where $k \geq 0$ and $k \leq n$,

when n is the length of the trace t being checked. This means, $k = 0$ corresponds to the judgment $\Phi_0 \vdash_\Delta \epsilon \rhd t : \mathit{trace}$. Likewise for $f_\Theta(k)$, i.e., the pair

$$f_\Delta(k), f_\Theta(k)$$

corresponds to the status of the two derivations for $\vdash_\Delta$ and $\vdash_\Theta$ at depth k, i.e., with a past of k labels left of the $\rhd$-separator in the corresponding judgments at that level.[9] By $l(k)$, we refer to the judgments of the linear derivation at distance k from the bottom of the linear derivation. For the induction step, we show incoming calls as one typical example.

Subcase: L-CALLI$^\Delta_{1,2}$ (and L-SKIPI$^\Delta$, resp. L-CALLI$^\Theta$ and L-SKIPI$^\Theta$): $a = \nu(\Phi').n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?$ and $\Phi \vdash r \rhd o_s \xrightarrow{a} o_r$.
By Lemma 5, there is exactly one instance of L-CALLI$^\Delta$ in the derivation from $f_\Delta(k)$ to $f_\Delta(k+1)$, all other steps (if any) are L-SKIPI$^\Delta$-steps. Now, the condition of the L-CALLI$^\Delta_{1,2}$ rule requires $\Delta_i \vdash o_s$, yielding one of the conditions of the corresponding call rule in the linear system.

Next the *typing* update in the linear system. By Lemma 1 we know:

$$\acute{\Phi}_\Delta = \bigoplus\nolimits_\Delta(\{\Phi_i\} + o_s \xrightarrow{a}) = (\bigoplus\nolimits_\Delta \Phi_i) + a$$

and analogously for the f_Θ-side

$$\acute{\Phi}_\Theta = \bigoplus\nolimits_\Theta(\{\Phi_j\} + \xrightarrow{a} o_r) = (\bigoplus\nolimits_\Theta \Phi_j) + a\ ,$$

and since furthermore $\{\Phi_i\}$ and $\{\Phi_j\}$ are consistent, $\acute{\Phi}_\Delta = \acute{\Phi}_\Theta$ (by the same Lemma 1).

For the *connectivity* update, we need to show that

$$\acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r$$

in the premise of L-CALLI$_{1,2}$, where o_s is the object determined above. By Lemma 2 we know that

$$\mathcal{L}(\{\Phi_i\}, \{\Phi_j\}) + o_s \xrightarrow{a} o_r = \mathcal{L}((\{\Phi_i\}, \{\Phi_j\}) + o_s \xrightarrow{a} o_r)\ ,$$

as required.

Remains the checks, which are covered by the f_Δ-part of the derivation. Well-typedness, as required by $\acute{\Phi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : ok$ of the call rule of the linear system is directly covered by L-CALLI$^\Delta_{1,2}$ and the straightforward weakening property of the type system. For the connectivity check $\acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : ok$, we know that $\acute{\Delta}_i \vdash o_s$ by the premise of the only instance of L-CALLI$^\Delta_{1,2}$ in the branching system in that step. By the *typing* premise in the same rule instance, we know $\acute{\Phi}_i \vdash \lfloor a \rfloor : ok$. To be well-typed, all names of $\lfloor a \rfloor$, the (free) core of a, must be covered by $\acute{\Phi}_i$. In the linear system with its explicit representation

[9] The length of the future may vary.

of connectivity this is interpreted as $\acute{\Delta}_i, \acute{\Sigma}_i; \acute{E}_{\Delta_i} \vdash o_1 \leftrightharpoons\hookrightarrow o_2 : \acute{\Theta}_i, \acute{\Sigma}_i$ for all names o_1 and o_2 where $\acute{\Delta}_i \vdash o_1$ and $\acute{\Delta}_i, \acute{\Sigma}_i, \acute{\Theta}_i \vdash o_2$. Note in particular that $\acute{\Delta}_i \vdash o_s$, which implies $\acute{\Delta}_i, \acute{\Sigma}_i; \acute{E}_{\Delta_i} \vdash o_s \leftrightharpoons\hookrightarrow \mathit{fn}(\lfloor a \rfloor) : \acute{\Theta}_i$, and furthermore in the overall context $\acute{\Delta}, \acute{\Sigma}; \acute{E}_{\Delta} \vdash o_s \leftrightharpoons\hookrightarrow \mathit{fn}(\lfloor a \rfloor) : \acute{\Theta}$, as required by the premise of L-CALLI$_{1,2}$ (cf. also Definition 2).

In case of L-CALLI$^{\Delta}_{0,3}$, the argument works analogously, where o is used instead of o_s in the updates, and where o is determined by by the premise $\Delta \vdash o$ of rule L-CALLI$^{\Delta}_{0,3}$.

Subcase: L-SPAWNI (and L-SKIPI)
Analogously.

Case: $\Leftarrow$
Analogously, as the Lemmas 1 and 2 work in both directions. □

3.4 Soundness of the abstractions

The section contains the basic soundness results of the abstractions.

With 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 14. *The judgment $\Xi \vdash C$ holds, if $\Delta, \Sigma \vdash C : \Theta, \Sigma$, and if $C \vdash n_1 \hookrightarrow n_2$, then $\Xi \vdash n_1 \leftrightharpoons\hookrightarrow n_2$.*

We simply write $\Xi \vdash C$ 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 15 (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 *sub-context* of Δ_1).

Lemma 7 (Subject reduction). *If $\Xi \vdash C \overset{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, then $\acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}$. A fortiori: If $\Delta, \Sigma, \Theta \vdash n : T$, then $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash n : T$.*

Proof. By induction on the number of reduction steps. That each internal step, structural congruence, and the external steps preserve well-typedness is shown by straightforward inspection of the rules, resp. induction. □

Besides the static abstraction of the type system, also the assertions about the heap topology (cf. Definition 14) preserved.

Lemma 8 (Soundness of the connectivity abstraction). *If* $\Xi \vdash C \stackrel{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$*, then* $\acute{\Xi} \vdash \acute{C}$*.*

An interesting invariant concerns the connectivity of names transmitted boundedly. Incoming communication, e.g., not only updates the commitment contexts —something one would expect— but also the *assumption* contexts. The fact that no new information is learnt about already known objects ("no surprise") in the assumptions can be phrased using the notion of conservative extension.

Definition 16 (Conservative extension). *For pairs* (Φ, E_Δ) *and* $(\acute{\Phi}, \acute{E}_\Delta)$ *of name context and connectivity context, i.e.,* $E_\Delta \subseteq \Phi \times \Phi$ *(and analogously for* $(\acute{\Phi}, \acute{E}_\Delta)$*), we write* $(\Phi, E_\Delta) \vdash (\acute{\Phi}, \acute{E}_\Delta)$ *if the following two conditions holds:*

1. $\acute{\Phi} \vdash \Phi$ *and*
2. $\acute{\Phi} \vdash n_1 \leftrightharpoons n_2$ *implies* $\Phi \vdash n_1 \leftrightharpoons n_2$*, for all* n_1, n_2 *with* $\Phi \vdash n_1, n_2$*.*

Lemma 9 (No surprise). *Let* $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$ *for some incoming label* a*. Then* $\Delta, \Sigma; E_\Delta \vdash \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta$*. For outgoing steps, the situation is dual.*

Proof. By definition of the incoming steps from Table 10 (resp. 11), using the context update from Definition 3 and 4. □

Lemma 10 (Soundness of legal trace system). *If* $\Delta_0; \vdash C : \Theta_0;$ *and* $\Delta_0; \vdash C : \Theta_0; \stackrel{t}{\Longrightarrow}$*, then* $\Delta_0 \vdash t : \mathit{trace}\, \Theta_0$*.*

Proof. The legal trace system of Table 14 (resp. its dual variant for Δ) can be re-formulated into an equivalent linear representation with connectivity contexts E_Δ and E_Θ as in the semantics (Lemma 6). The checks and updates of the assumption contexts then match exactly the checks and updates of the external steps. The only additional provisos of the legal traces stipulate that the calls and returns are parenthetic (i.e., each thread must be "balanced"). It is straightforward to check, that the operational semantics allows only prefixes of balanced traces. □

4 Conclusion

Related work In [17] a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* [18,16] specification language is presented. It is based on a refinement of the simple trace semantics called the complete-readiness model, which is related to the readiness model of Olderog and Hoare [15]. In [19], full

abstraction in an object calculus with subtyping is investigated. The setting is slightly different from the one 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 [14] extended their work [13] on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. However, their semantics avoids object connectivity by using a notion of *package*. [9] tackles full abstraction and observable component behavior and connectivity in a UML-setting.

Future work We plan to extend the language with further features to make it more resembling *Java* or $C^{\#}$. Concerning the concurrency model, objects should be extended by lock-*synchronization* as provided by *Java*'s `synchronized` methods, and by wait- and signal-methods. A preliminary study in this direction is [5]. Another interesting direction for extension concerns the type system, in particular to include *subtyping* and *inheritance*. This is challenging especially if the component may inherit from environment classes and vice versa. For a first step in this direction we will concentrate on subtyping alone, i.e., relax the type discipline of the calculus to subtype polymorphism, but without inheritance. Another direction is to extend the semantics to a *compositional* one; currently, the semantics is open in that it is defined in the context of an environment. However, general composition of open program fragments is not defined. Finally, we work on adapting the full abstraction proof of [3] to the new setting, i.e., to deal with thread classes. The results of Section 3.4 are covering the soundness-part of the full-abstraction result.

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 and K. Araki, editors, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 37–51. 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. [8], pages 296–316.
5. E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. Jan. 2006. Submitted as conference contribution.
6. E. Ábrahám, A. Grüner, and M. Steffen. Dynamic heap-abstraction for open, object-oriented systems with thread classes. Technical Report 0601, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Jan. 2006.

7. P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
8. 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)*, volume 3657 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
9. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [8], pages 49–69.
10. ECMA International Standardizing Information and Communication Systems. *C# Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.
11. 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.
12. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.
13. 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.
14. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In M. Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.
15. 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.
16. B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Series in Computer Science. Prentice Hall, 1990.
17. G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
18. J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 1989.
19. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.