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

June 24, 2005

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

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

Abstract. 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.
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. We show basic soundness results of the abstraction.

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

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 assumed 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

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

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 very language and its semantics.

With these goals in mind, this paper deals primarily with the following three features, which correspond to those of modern class-based object-oriented languages like *Java* [10] or $C^{\#}$ [8] 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 mentioned languages feature concurrency based on *threads* (as opposed to processes or active objects).
- *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 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;
- finally 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 groups of objects may merge.

In [3,4] we showed that the last point, namely 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.

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 [5].

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 [9] and in particular [11]. Compared to our previous work for instance in [2], we added thread classes as generators of activity.

2.1 Syntax

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

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

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

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

The available types include *thread* as the type of threads. Furthermore, objects are typed by the name of their class. As auxiliary types we have $T_1 \times \ldots \times T_k \to T$ as the type of methods as well as for thread classes (in which case the result type T equals *thread*), and furthermore $[l_1{:}U_1, \ldots, l_k{:}U_k]$ as the type or interface of unnamed objects, and $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$ as the type for classes.

2.2 Operational semantics

For the operational semantics, we concentrate on the interface behavior. For want of space, we omit the (straightforward) definitions of the component-internal steps, for instance-internal method calls or internal thread creation. For the definition of the semantics, we refer to [5].

The external steps define the interaction of the component with the environment. In particular, the semantics is defined in reference to assumption and

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

Table 1. Abstract syntax

commitment contexts. The static part of the contexts corresponds to the static type system (we again refer to [5] for the full definition) and takes care that, e.g., only well-typed values are received from the environment. The contexts, however, need to contain also a *dynamic* part dealing with the potential *connectivity* of objects and thread names and which corresponds to an abstraction of the heap of the program.

A component exchanges information with the environment via *calls, returns,* and *spawn* actions (cf. Table 2). In the call and return labels, the mentioned n is the active thread that issues the call or returns from the call. In the thread instantiation label, n is the name of the new thread; the thread which spawned the new thread is *not* part of the label.[3] Furthermore note that there are no separate external labels for object instantiation: Externally instantiated objects are created only at the point when they are actually accessed for the first time, which we call "*lazy instantiation*". Given a label $\nu(\Phi).\gamma'$ where Φ is a name context, i.e., a sequence of single $\nu(n{:}T)$ bindings and where γ' does not contain any binders, we call γ' the *core* of the label. Given a label γ, we refer with $\lfloor\gamma\rfloor$ to its core. Analogously for send and receive labels.

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

Table 2. Labels

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

2.2.1 Connectivity contexts In the presence of cross-border instantiation, the semantics must contain a representation of the connectivity, which will be formalized by a relation on the names of the calculus and which can be seen as an abstraction of the program's heap; for the exact definition, see Equation (2) and (3) below. 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*. The assumptions consist of a part Δ, Σ concerning the existence (plus static typing information) of *named entities* in the environment. The semantics maintains as invariant that the assumption and commitment contexts are disjoint concerning object and class names, whereas a thread name occurs as assumption iff. it is mentioned in the commitments. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names. This means, as invariant we maintain for all judgments $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ that Δ, Σ, and Θ are pairwise disjoint.

The semantics must book-keep which objects of the environment have been told which identities. This means it must take 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, i.e., those from Θ. In analogy to the name contexts Δ and Θ, E_Δ (the connectivity context) expresses assumptions about the environment, and E_Θ commitments of the component:

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

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Sigma + \Delta)$. Since in the language we allow the sending of thread names, we must include pairs from $\Delta \times \Sigma$ (resp. $\Theta \times \Sigma$) into the connectivity. We write $o \hookrightarrow n$ ("o may know n") for pairs from the relations E_Δ, resp. 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, i.e.,

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

where $\hookrightarrow\!\downarrow_\Delta$ is the projection of $\hookrightarrow$ to Δ. We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Sigma + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\hookrightarrow$ for that union. As judgment, we use $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta, \Sigma$, resp. $\Delta, \Sigma; E_\Delta \vdash o \leftrightharpoons\!\hookrightarrow n : \Theta, \Sigma$. For Θ, Σ, E_Θ, and Δ, Σ, the definitions are applied dually.

The relation $\leftrightharpoons$ partitions the objects from Δ (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 communication, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. Incoming new names, exchanged boundedly, however, update both commitments and assumptions.

Remark 1 (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.*

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.

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

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

The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1.

Furthermore, we augment the syntax of the method definitions accordingly, such that each method call and each spawn step is preceded by an annotation of the caller; i.e., instead of $\varsigma(\mathit{self}{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots \mathit{spawn}\, c_t(\vec{z})\ldots)$ we write

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

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

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

for each mentioned call. When the thread is instantiated, $\odot$ is replaced by $\odot_n$ where n is the identity of the new thread. Given the above thread class, we denote by $c_t(\vec{v})$ the replacement $t[\odot_n, \vec{v}/\odot, \vec{x}]$, when t is the body of the thread class definition. The initial thread, which is not instantiated from a thread class but given directly (in case the activity starts in the component) starts with $\odot_n$ as augmentation, if the initial thread is named n. If the component is renamed

by α-conversion, n and $\odot_n$ are renamed simultaneously. The steps of the internal semantics must be adapted accordingly. We also omit the typing rules for the augmentation, as they are straightforward.

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

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

The assumption context $\Delta, \Sigma; E_\Delta$ can be seen as an abstraction of the (not-present) environment; more precisely, it represents the potential behavior of all possible environments.

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

The check whether the current assumptions are met in an incoming communication step is given in Definition 1.

Definition 1 (Connectivity check). *An incoming core label a with sender o_s and receiver o_r is* well-connected *wrt. an assumption-commitment context $\acute{\Xi}$ (written $\acute{\Xi} \vdash o_s \xrightarrow{a} o_r : ok$) if:*

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

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

Besides *checking* whether the connectivity assumptions are met before a transition, the contexts are *updated* by a step, reflecting the change of knowledge.

Definition 2 (Name context update: Φ+a). *The* update $\acute{\Phi}$ *of an assumption-commitment context Φ wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor$ is defined as follows.*

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

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

Definition 3 (Connectivity context update). *The update* $(\acute{E}_\Delta, \acute{E}_\Theta)$ *of an assumption-commitment context* (E_Δ, E_Θ) *wrt. an incoming label* $a = \nu(\Phi')\lfloor a \rfloor?$ *with sender* o_s *and receiver* o_r *is defined as follows.*

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

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

Combining Definitions 2 and 3, we write $\Xi + o_s \xrightarrow{a} o_r$ when updating the name and the connectivity at the same time.

Besides Definition 1, which checks whether the connectivity assumptions are met for the label at hand, we must additionally check the *static* assumptions, i.e., whether the transmitted values are of the correct types. In slight abuse of notation, we write $\Delta, \Sigma, \Theta \vdash o_s \xrightarrow{a} o_r : T$ for that check, where T is type of the expression in the program that gives rise to the label. We omit the exact definition here which can be found in [5]. We combine the connectivity check of Definition 1 and the type check notationally into one single judgment $\Xi \vdash o_s \xrightarrow{a} o_r : T$.

2.2.4 Operational rules With all the ancillary definitions at hand, we can define the operational rules of the semantics (cf. Table 3).

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

The three rules for incoming calls deal with three different situations as to when an incoming call may happen: A reentrant call[4], a call of thread where the thread name is already known in the component, and a call of a thread which is new to the component.

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

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

For reentrant method calls (cf. rule CALLI_1), the thread is blocked, i.e., it has left the component previously via an outgoing call. The object that had been the target of the call is remembered as part of the augmented block syntax. In the rule it is referred to as o_s, as it represents the sender's clique of the current incoming call.

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

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

The last call rule CALLI$_3$ deals with the situation, that the thread n enters the component for the first time. This is assured by the premise $\Sigma' \vdash n : thread$. As in CALLI$_2$, we do not have an indication from which clique the call originates, since the corresponding thread is *new.* What is assured is that the new thread has been created at some point before as instance of some environment thread class —otherwise the cross-border instantiation would have been observed and the thread name would not be fresh now— and by some environment clique. Indeed, *any* existing environment clique is a candidate that might have created the thread n. So the update to $\acute{\Xi}$ *non-deterministically guesses* to which environment clique the thread's origin $\odot_n$ belongs to. Note that $\odot_{\Sigma'}$ contains $\odot_n$ since $\Sigma' \vdash n$, which means $\acute{\Delta} \vdash \odot_n$ after the call.

For incoming thread creation in rule SPAWNI, we need again to know the origin of the call, i.e., the spawning clique. The situation is similar to the one for CALLI$_3$, in that the origin of the communication needs to be guessed. In the case of CALLI$_3$, we use $\odot_n$ 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 2 and 3) 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.

The remaining rules deal with outgoing communication and are simpler, as the "check-part" is omitted: With the code of the program present, the checks are guaranteed to be satisfied.

In addition to the external steps of Table 3, there are similar ones for communication via returns, and rules dealing with initial steps. They are included in the technical report [5].

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

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

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

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

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

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

Table 3. External steps

3 Legal traces

In this section we present a proof system which provides an independent characterization of which traces are possible as interface behavior between component and environment. We call those traces legal. "Half" of the work has been done already by the careful design of the open semantics of Section 2.2.4, where the absent environment is represented abstractly by the name and connectivity contexts. For characterizing the legal traces, we analogously abstract away from the program code, which makes the system completely symmetric. Remember that the assumption and commitment contexts in the operational semantics where used asymmetrically insofar, as the commitment contexts where updated as overapproximation of the actual component, but not used in *checking* whether

$$\frac{\begin{array}{c} a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \vdash r \rhd o_s \xrightarrow{a} o_r \\ \acute{\Xi} = \bigoplus \Xi_i + a \quad \Theta_i \vdash o_r \quad \epsilon \neq a_i = (a, o_r) \downarrow_{\Theta_i} \quad \acute{\Delta}, \acute{\Sigma} \vdash r\ a \rhd s : trace\, \acute{\Theta}, \acute{\Sigma} \end{array}}{\Delta_1, \Sigma_1 \vdash r \rhd a_1\ s : trace\, \Theta_1, \Sigma_1 \quad \ldots \quad \Delta_k, \Sigma_k \vdash r \rhd a_k\ s : trace\, \Theta_k, \Sigma_k}\ \text{L-CALLI}$$

$$\frac{\Delta, \Sigma \vdash ra \rhd s : trace\, \Theta, \Sigma \quad a = \gamma? \quad \vdash r \rhd o_s \xrightarrow{a} o_r \quad \Theta \not\Vdash o_r}{\Delta, \Sigma \vdash r \rhd s : trace\, \Theta, \Sigma}\ \text{L-SKIPI}$$

Table 4. Legal traces, branching on Θ

a component step, i.e., an outgoing communication, is possible as next interaction.

3.1 A branching derivation system characterizing legal traces

Unlike the treatment in the operational semantics, 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 tree representing a component, respectively environment clique at the end. The judgments are of the form

$$\Delta, \Sigma \vdash_\Theta r \rhd s : trace\, \Theta, \Sigma \tag{5}$$

where r represents the history or past interaction, 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 on the commitment side, 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 object connectivity, i.e., the acquaintance of the clique represented by Θ with objects of the environment. This information 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 implies that the system of Table 4 *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.

Now to the rules of Table 4. As before, rule L-CALLI deals with incoming calls. The call is possible only when the thread is input call enabled after the current history. This is checked by the premise $\vdash r \rhd o_s \xrightarrow{a} o_r : ok$, which also determines caller and callee. We omit the definition of $\vdash r \rhd o_s \xrightarrow{a} o_r : ok$, characterizing enabledness of a after trace r. The definition is the straightforward extension of the one from [4] to a multi-threaded setting.

Since from the perspective of the component, the connectivity of the environment is no longer represented as assumption, there are *no* premises checking connectivity! An interesting part concerns the treatment of the commitment context: 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 tree (respectively forest) correspond to freshly created cliques.

In rule 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 the communication, Θ is the disjoint combination of the Θ_i's plus the lazily instantiated objects from Θ'. For the cross-border connectivity, i.e., the environmental objects known by the component cliques, 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.[5] These restrictions are covered by the definition of the (partial) operation $\bigoplus \Xi_i$.

We omit the rules dealing with incoming returns and incoming spawn labels, and furthermore those for outgoing communication.

The skip-rules stipulate that an action a which does not belong to the component clique under consideration, is 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).

Definition 4 (Legal traces, branching system). *We write $\Delta \vdash_\Theta t : trace\,\Theta$, if there exists a derivation forest using the rules of Table 4 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* L-CalliI$_0$ *or* L-CallO$_0$. *Using the dual rules, we write $\vdash_\Delta$ instead of $\vdash_\Theta$.*

We write $\Delta \vdash_{\Delta \wedge \Theta} t : trace\,\Theta$, if there exits a pair of derivations in the $\vdash_\Delta$- and the $\vdash_\Theta$- system with a consistent pair of root judgments.

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

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash \lfloor a \rfloor : ok \qquad \Delta \not\vdash static \quad \acute{\Delta} \vdash o_s \quad \acute{\Xi} = \Xi + a \quad \acute{\Delta}, \acute{\Sigma} \vdash r\ a \rhd s : trace\,\acute{\Theta}, \acute{\Sigma}}{\Delta, \Sigma \vdash r \rhd a\ s : trace\,\Theta, \Sigma}\ \text{L-CalliI}$$

$$\frac{\acute{\Delta} = \Delta \not\vdash o_s \quad \Delta, \Sigma \vdash ra \rhd s : trace\,\Theta, \Sigma \quad a = \gamma? \quad \vdash r \rhd o_s \xrightarrow{a} o_r}{\Delta, \Sigma \vdash r \rhd s : trace\,\Theta, \Sigma}\ \text{L-SkipI}$$

Table 5. Legal traces, branching on Δ

To accommodate for the simpler structure of the contexts, we adopt the notational conventions (cf. Notation 1) appropriately.

The way a communication step updates the name context can be defined as simplification of the treatment in the operational semantics (cf. Definition 2). As before we write $\Phi + a$ for the update.

3.2 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? 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 5. *The judgment $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ holds, if $\Delta, \Sigma \vdash C : \Theta, \Sigma$, and if $C \vdash n_1 \hookrightarrow n_2$, then $\Theta, \Sigma; E_\Theta \vdash n_1 \leftrightharpoons n_2 : \Delta, \Sigma$.*

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

Lemma 1 (Subject reduction). *$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$, then $\acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}$. A fortiori: If $\Delta, \Sigma, \Theta \vdash n : T$, then $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash n : T$.*

Besides the static abstraction of the type system, also the assertions about the heap topology (cf. Definition 5) preserved.

Lemma 2 (Soundness of the connectivity abstraction). *$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$, then $\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$.*

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 6 (Conservative extension). *Given two 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 3 (No surprise). *Let $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \stackrel{a}{\rightarrow} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$ 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.*

Lemma 4 (Soundness of legal trace system). *If $\Delta_0; \vdash C : \Theta_0;$ and $\Delta_0; \vdash C : \Theta_0; \stackrel{t}{\Longrightarrow}$, then $\Delta_0 \vdash t : \mathit{trace}\, \Theta_0$.*

4 Conclusion

Related work [13] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* specification language. 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. [14] investigates full abstraction in an object calculus with subtyping. 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 [12] extended their work [11] on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*. [7] tackles the problem of 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 furthermore monitor synchronization via wait- and signal-methods. 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 3.2 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, editor, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, July 2004.
4. E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In Bosangue et al. [6]. To appear.

5. E. Ábrahám, A. Grüner, and M. Steffen. An open structural operational semantics for an object-oriented calculus with thread classes. Technical Report 0505, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2005.
6. M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors. *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
7. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [6]. To appear.
8. ECMA International Standardizing Information and Communication Systems. *C# Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.
9. 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.
10. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.
11. 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.
12. 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.
13. G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
14. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.