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

March 30, 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. 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

1 Introduction

An *open* system is a program fragment or component interacting with its environment. 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, a characterization of the 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 with languages supporting:

- *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.

* 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).

- *concurrency:* the languages feature concurrency based on *threads* and *thread classes* (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.*

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 due to object creation and tree-structured in that previously separate object groups may merge.

In [1,2] 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".* 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 [1,2] with *thread classes* and allow to communicate thread names. This requires to consider cross-border *activity* generation and 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: The branching nature of the heap abstraction—connected object groups may merge by communication—is reflected in the branching structure of the derivation system. Finally, we show soundness of the abstractions.

Overview The paper is organized as follows. Section 2 defines syntax and semantics of the calculus. Section 3 characterizes the observable behavior of an open system and presents the soundness results. Related and future work is discussed in Section 4. See [3] for a full description of semantics and type system.

2 A multi-threaded calculus with thread classes

2.1 Syntax

The abstract syntax is given in Table 1. For names, we will generally use o and its syntactic variants for objects, c for classes (in particular c_t for thread classes), and n when being unspecific. A class $c[\![O]\!]$ with name c defines its methods and fields. A method $\varsigma(self{:}c).t_a$ provides the method body abstracted over the ς-bound "self" and the formal parameters. An object $o[c, F]$ of type c stores the current field values. We use l for fields, $l = f$ for field declaration, field access is written as $v.l$, and field update as $v.l := v'$. *Thread classes* $c_t\langle\!\langle t_a \rangle\!\rangle$ with name c_t

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

carry their abstract code in t_a. A thread $n\langle t\rangle$ with name n is basically either a value or a sequence of expressions, notably method calls $v.l(\vec{v})$, object creation *new* c, and *thread instantiation* *spawn* $c_t(\vec{v})$.

As types we have *thread* for threads, class names c as object types, $T_1 \times \ldots \times T_k \to T$ as the type of methods and thread classes (in last case T equals *thread*), $[l_1{:}U_1, \ldots, l_k{:}U_k]$ for unnamed objects, and $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$ for classes.

2.2 Operational semantics

For lack of space we concentrate on the interface behavior and omit the definitions of the component-internal steps like internal method calls [3].

The external steps define the interaction between component and environment in an assumption-commitment context. The static part of the context corresponds to the static type system [3] 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.

The component-environment interaction is represented by labels a:

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

For call and return, n is the active thread. For spawning, n is the new thread. There are no labels for object creation: Externally instantiated objects are created only when they are accessed for the first time ("*lazy instantiation*"). For labels $a = \nu(\Phi).\gamma?$ or $a = \nu(\Phi).\gamma!$ with Φ a sequence of $\nu(n{:}T)$ bindings and γ does not contain any binders, $\lfloor a \rfloor = \gamma$ is the *core* of the label a.

2.2.1 Connectivity contexts In the presence of cross-border instantiation, the semantics must contain a representation of the connectivity, which is formalized by a relation on names and which can be seen as an abstraction of the

program's heap; see Eq. (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 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. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names. The semantics maintains as invariant that 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 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, 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 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 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. 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. 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.

If a thread is instantiated *without* connection to any object, like the initial thread, we need a syntactical representation $\odot_n$ for the clique the thread n starts in. If the single initial thread starts within the component, the contexts of the initial configuration $\Delta_0 \vdash C : \Theta_0$ assert $\Theta_0 \vdash \odot$. Otherwise, $\Delta_0 \vdash \odot$.

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 update both E_Θ and E_Δ.

2.2.2 Augmentation We extend the syntax by two auxiliary expressions o_1 *blocks for* o_2 and o_2 *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

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

instead of $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots spawn\, c_t(\vec{z})\ldots)$. Thread classes are augmented by $\odot$ instead of *self*. 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 $t[\odot_n, \vec{v}/\odot, \vec{x}]$. The initial thread n_0, which is not instantiated but is given directly (in case it starts in the component), has $\odot_{n_0}$ as augmentation. We omit the adaptation of the internal semantics and the typing rules for the augmentation.

2.2.3 Use and change of contexts

Notation 1 *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 formulated as 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 environment, as it represents the potential behavior of all possible environments. The check whether the current assumptions are met in an incoming communication step is given in Definition 1. Note that in case of an incoming *call* label, $fn(a)$, the free names in a, includes the receiver o_r and the thread name.

Definition 1 (Connectivity check). *An incoming core label* a *with sender* o_s *and receiver* o_r *is* well-connected *wrt.* $\acute{\Xi}$ *(written* $\acute{\Xi} \vdash o_s \xrightarrow{a} o_r : ok$*) if* $\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash o_s \leftrightharpoons\hookrightarrow fn(a) : \acute{\Theta}, \acute{\Sigma}$*.*

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: $\Phi + a$). *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. $\acute{\Theta} = \Theta + \Theta'$*. For spawning,* $\acute{\Theta} = \Theta + (\Theta', \odot_n)$ *with* n *the spawned thread.*
2. $\acute{\Delta} = \Delta + (\odot_{\Sigma'}, \Delta')$*. For spawning of thread* n*,* $\odot_{\Sigma' \setminus n}$ *is used instead of* $\odot_{\Sigma'}$*.*
3. $\acute{\Sigma} = \Sigma + \Sigma'$*.*

The notation $\odot_{\Sigma'}$ *abbreviates* $\odot_n$ *for all thread identities from* 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*).*

Definition 3 (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 defined as follows.*

1. $\acute{E}_\Theta = E_\Theta + o_r \hookrightarrow \mathit{fn}(\lfloor a \rfloor)$.
2. $\acute{E}_\Delta = E_\Delta + o_s \hookrightarrow \Phi', \odot_{\Sigma'}$. *For spawning of* n, $\odot_{\Sigma' \setminus n}$ *is used instead of* $\odot_{\Sigma'}$.

Combining Definitions 2 and 3, 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 1, we must also check the *static* assumptions, i.e., whether the transmitted values are of the correct types. In slight abuse of notation, we write $\Phi \vdash o_s \xrightarrow{a} o_r : T$ for that check, where T is type of the expression in the program that gives rise to the label (see [3] for the definition). We combine the connectivity check of Definition 1 and the type check into a single judgment $\Xi \vdash o_s \xrightarrow{a} o_r : T$.

2.2.4 Operational rules Three CallI-rules for incoming calls deal with three different situations: 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 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.

To deal with component entities (threads and objects) that are being created during the call, $C(\Theta', \Sigma')$ stands for $C(\Theta') \parallel C(\Sigma')$, 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.

In CallI_2, the thread is not in the component, but the thread's name is already known. 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.

In CallI_3 a new thread n enters the component for the first time, as assured by $\Sigma' \vdash n : \mathit{thread}$. The new thread must be an instance of an environment thread class created by an environment clique, otherwise the cross-border instantiation would have been observed and the thread name would not be fresh. Since *any* existing environment clique is a candidate, the update to $\acute{\Xi}$ *non-deterministically*

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

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

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

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

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

Table 2. External steps

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 the situation is similar to CalliI$_3$, in that the spawner needs to be guessed. The last rule deals with outgoing call and is simpler, as the "check-part" is omitted: With the code of the program present, the checks are guaranteed to be satisfied. In Table 2 we omitted the rules for outgoing spawning, for returns, and for the initial steps [3].

3 Legal traces

Next we present an independent characterization of the possible interface behavior. "Half" of the work has been already done by the abstractly represented environment. For the legal traces, we analogously abstract away from the component, making the system completely symmetric.

3.1 A branching derivation system characterizing legal traces

Instead of connectivity contexts, now the *tree structure* of the derivation represents the connectivity and its change. There are two variants of the derivation system, one from the perspective of the *component,* and one for the *environment.* Each derivation corresponds to a *forest,* with each tree representing a

component resp. environment clique. In judgments $\Delta, \Sigma \vdash_\Theta r \rhd s : \mathit{trace}\,\Theta, \Sigma$, 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 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 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 3 *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.

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

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

Table 3. Legal traces, branching on Θ (incoming call and skip)

In L-CALLI of Table 3, the incoming call is possible only when the thread is input call enabled after the current history. This is checked by the premise $\Phi \vdash r \rhd o_s \xrightarrow{a} o_r$, which also determines caller and callee. As from the perspective of the component, the connectivity of the environment is no longer represented as assumption, there are *no* premises checking connectivity! Interesting is 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 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, 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. These restrictions are covered by $\bigoplus \Xi_i$.

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). We omit the remaining rules (see [3]).

Definition 4 (Legal traces, tree system). *We write $\Delta \vdash_\Theta t : \mathit{trace}\,\Theta$, if there exists a derivation forest using the rules of Table 3 with roots $\Delta_i, \Sigma_i \vdash t \rhd \epsilon : \mathit{trace}\,\Theta_i, \Sigma_i$ and a leaf justified by one of the initial rules* L-CALLI$_0$ *or* L-CALLO$_0$*. Using the dual rules, we write $\vdash_\Delta$ instead of $\vdash_\Theta$. We write $\Delta \vdash_{\Delta\wedge\Theta} t : \mathit{trace}\,\Theta$, if there exits a pair of derivations in the $\vdash_\Delta$- and the $\vdash_\Theta$- system with a consistent pair of root judgments.*

To accommodate for the simpler context structures, we adapt 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

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 $\Theta, \Sigma; E_\Theta \vdash n_1 \leftrightharpoons\hookrightarrow n_2 : \Delta, \Sigma$.*

We 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). *Assume $\Xi \vdash C \stackrel{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$. Then*

1. *$\acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}$. A fortiori: If $\Delta, \Sigma, \Theta \vdash n : T$, then $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash n : T$.*
2. *$\acute{\Xi} \vdash \acute{C}$.*

Definition 6 (Conservative extension). *Given 2 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 2 (No surprise). *Let $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \stackrel{a}{\longrightarrow} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$ 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 3 (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 [8] 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. In [9], 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 [7] extended their work [6] 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*. [5] 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 $C^{\#}$. Besides *monitor synchronization* using object locks and wait and signal methods, as provided by *Java*, another interesting direction concerns *subtyping* and *inheritance.* This is challenging especially if the component may inherit from environment classes and vice versa. Another direction is to extend the semantics to a *compositional* one. Finally, we work on adapting the full abstraction proof of [1] to deal with thread classes. The results of Section 3.2 are covering the soundness-part of the full-abstraction result.

References

1. 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 *LNCS*, pages 37–51. Springer-Verlag, July 2004.
2. 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. [4], pages 296–316.
3. 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.
4. M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors. *Proceedings of FMCO 2004*, volume 3657 of *LNCS*. Springer-Verlag, 2005.
5. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [4], pages 49–69.
6. 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.
7. 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 *LNCS*, pages 423–438. Springer-Verlag, 2005.
8. G. P. Smith. *An Object-Oriented Approach to Formal Specification.* PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
9. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.