Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes*

— Extended Abstract —

December 6, 2004

Erika Ábrahám[2], Marcello M. Bonsangue[3],
Frank S. de Boer[4], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] University Freiburg, Germany
[3] University Leiden, The Netherlands
[4] CWI Amsterdam, The Netherlands

Abstract. The concurrent object calculus has been investigated as a core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects. The combination of this form of concurrency with objects corresponds to features known from the popular language *Java*. One distinctive feature, however, of the concurrent object calculus is that it is *object-based*, whereas the mainstream of object-oriented languages is *class-based*.
This work explores the semantical consequences of introducing classes to the calculus. Considering classes as part of a component makes instantiation a possible interaction between component and environment. A striking consequence is that to characterize the observable behavior we must take *connectivity* information into account, i.e., the way objects may have knowledge of each other. In particular, unconnected environment objects can neither determine the absolute order of interaction and furthermore cannot exchange information to compare object identities.
We formulate an operational semantics that incorporates the connectivity information into the scoping mechanism of the calculus. As instantiation itself is unobservable, objects are instantiated only when accessed for the first time ("*lazy instantiation*").
Furthermore we use a corresponding trace semantics for full abstraction wrt. a may-testing based notion of observability.
Keywords: multithreading, class-based object-oriented languages, formal semantics, full abstraction

1 Introduction

The notion of component is well-advertised as structuring concept for software development. Even if there is not too much agreement about what constitutes a

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

component in concrete software engineering terms, one aspect should go undisputed: At the bottom line, a component means a "program fragment" being *composed,* which raises the question what the semantics of a component is. A natural approach is to take an observational point of view: two components are observably equivalent, when no observing context can tell them apart.

In the context of concurrent, *object-based* programs and starting from may-testing as a simple notion of observation, Jeffrey and Rathke [7] provide a fully abstract trace semantics for the language. Their result roughly states that, given a component as a set of objects and threads, the fully abstract semantics consists of the set of traces at the boundary of the component, where the traces record incoming and outgoing calls and returns. At this level, the result is as one would expect, since intuitively in the chosen setting, the only possible way to observe something about a set of objects and threads is by exchanging messages.

The result in [7] is developed within the concurrent object calculus [5], an extension of the sequential ν-calculus [10] which stands in the tradition of various object calculi [1] and also of the π-calculus [9,11]. The chosen language has been proposed as core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects, but distinctive feature is that it is *object-based,* which in particular means that there are no *classes* as templates for new objects. This is in contrast to the mainstream of object-oriented languages where the code is organized in classes, one well-known example being *Java*. This work addresses therefore the following question:

> What changes when switching from an object-based to a class-based setting, a setting which corresponds to features as found in a language like multithreaded *Java* or $C^{\#}$?

Considering the observable behavior of a component, we have to take into account that in addition to objects, which are the passive entities containing the instance state, and threads, which are the active entities, *classes* come into play. Classes serve as a blueprint for their instances and can be conceptually understood as particular objects supporting just a method which allows to generate instances. Indeed, ultimately, the observer consists only of classes since the program code is structured into classes, and objects exist only at run-time.

Crucial in our context is that now the division between the program fragment under observation and its environment also separates *classes:* There are classes internal to the component and those belonging to the environment. As a consequence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well. This possibility of *cross-border instantiation* is absent in the object-based setting: Objects are created by directly providing the code of their implementation, not referring to the name of a class, which means that the component creates only component-objects and dually the environment only environment objects.

To understand the bearing of this change on what is observable, we concentrate on the issue of instantiation across the demarcation line between component and its environment. The environment is considered as the observing

context which tries to determine the behavior of the component or program under observation. So imagine that the component creates an instance of an environment class, and the first question is: does this yield a component object or an environment object? As the code of the object is in the hand of the observer, namely being provided by the external class, the further interaction between the component and the newly created object can lead to observable effects and must thus be part of the behavior at the component's interface. In other words, instances of environment classes belong to the environment, and dually those of internal classes to the component.

To obtain a semantics which is abstract enough, it is crucial not just to cover all possible interface behavior —there is little doubt that sequences of calls, returns, and instantiations with enough information at the labels would do— but to capture it *exactly,* i.e., to exclude impossible environment interaction. As an obvious example: two consecutive calls from the same thread without outgoing communication in between cannot be part of the component behavior.

Whereas in the above situation, the object is instantiated to be part of the environment, the *reference* to it is kept at the creator, for the time being. So in case an object of the program, say o_1 instantiates two objects o_2 and o_3 of the environment, the situation informally looks as shown in Figure 1, where the dotted bubbles indicate the scope of o_2, respectively of o_3.

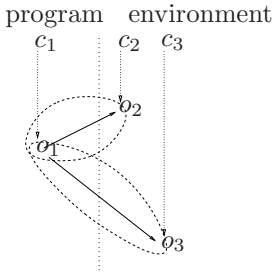

Fig. 1. Instances of external classes

In this situation, an incoming call from the environment carrying both names o_2 and o_3 is impossible, as the only entity aware of both references is o_1. Unless the component gives away the references to the environment, o_2 and o_3 are and remain completely separated.

Thus, to exclude impossible combinations of object references in the communication labels, the component must keep track of which objects of the environment are connected. The component has, of course, by no means full information about the complete system; after all it can at most trace what happens at the interface, and the objects of the environment can exchange information "behind the component's back". Therefore, the component must conservatively overapproximate the potential knowledge of objects in the environment, i.e., it must make *worst-case assumptions* concerning the proliferation of knowledge, which means it must assume that

1. once a name is out, it is never forgotten, and
2. if there is a possibility that a name is leaked from one environment object to another, this will happen.

Sets of environment objects which can possibly be in contact with each other form therefore equivalence classes of names —we call them *cliques*— and the formulation of the semantics must contain a representation of them. New cliques

can be created, as new objects can be instantiated without contact to others, and furthermore cliques can merge, if the component leaks the identity of a member of one clique to a member of another.

This paper investigates a class-based variant of the object calculus, formalizing the ideas sketched above about cliques of objects. Instantiation itself, even across the environment-program boundary, is unobservable, since the calculus does not have constructor methods. In the semantics, an externally instantiated object is created only at the point when it is actually accessed the for the first time, which we call "*lazy instantiation*". For want of space, we concentrate here on the intuition and stress the differences to the object-based setting. For deeper coverage we refer to the technical reports [2] and [3].

The paper is organized as follows. Section 2 contains the syntax of the calculus in which the result is presented, and a sketch of its semantics. In particular, the notions of lazy instantiation and connectivity of objects are formalized. Afterwards, Section 3 elaborates on the trace semantics, Section 4 fixes the notion of observability, and Section 5 states the full abstraction result. Finally in Section 6, we discuss related work.

2 A concurrent class calculus

In this section, we present the calculus used in our development. As we concentrate on the semantical issues of connectivity of objects and the interface behavior of a component, we only sketch the syntax, ignore typing issues and also omit structural equivalence rules, as they are rather standard. As mentioned, the reader will find details in the accompanying technical report.

The calculus is a syntactic extension of the concurrent object calculus from [5,7]. The basic change is the introduction of *classes,* where a class is a named collection of methods. In contrast to object references, class names are literals introduced when defining the class; they may be hidden using the ν-binder but unlike object names, the scopes for class names are *static.* Object names, on the other hand, are first-order citizens of the calculus in that they can be stored in variables, passed to other objects as method parameters, making the scoping *dynamic,* and especially they can be created freshly by instantiating a class.

A program is given by a collection of classes. A class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields. 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 $\varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_n{:}T_k).t$ 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 can contain as active entities *named threads* $n\langle t\rangle$, which, like objects, can be dynamically created. Unlike objects, threads are not instantiated by some statically named entity (a "thread class" as in *Java*), but directly created by providing the code. A thread basically is either a value (especially a reference to another named entity) or a sequence of expressions, notably method calls

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

Table 1. Abstract syntax

(written $o.l(\vec{v})$) and creation of new objects and new threads ($\mathit{new}\, c$ and $\mathit{new}\langle t\rangle$ where c is a class name and t a thread). We will generally use n and its syntactic variants as name for threads (or just in general for names), o for objects, and c for classes. Furthermore we will use f specifically for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update[5] as $f.x := v$.

Concerning the *operational semantics* of the calculus, the basic steps are mainly given in two levels: *internal* steps whose effect is confined within a component, and those with external effect. Interested mainly in the external behavior we elide the definition of the internal steps.

The *external* behavior of a component is given in terms of labeled transitions describing the communication at the interface of an *open* program. For the completeness of the semantics, it is crucial ultimately to consider only communication traces realizable by an actual program context which, together with the component, yields a well-typed closed program.

The concentration on actually realizable traces has various aspects, e.g., the transmitted values need to adhere to the static typing assumptions, only publicly known objects can be called from the outside, and the like. Being concerned with the dynamic relationship among objects, we omit also these aspects here. Besides that, this part is rather standard and also quite similar to the one in [7].

2.1 Connectivity contexts and cliques

The informal discussion in the introduction argued that in the presence of internal and external classes and cross-border instantiation, the component must *keep track* of which identities it gives away to which objects in order to exclude impossible behavior as described for instance in connection with Figure 1. The external semantics is formalized as labeled transitions between judgments of the

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

form

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

where $\Delta; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta; E_\Theta$ the *commitments*; alternative names are the required and the provided interface of the component. The assumptions consist of a part Δ concerning the existence (plus static typing information) of *named entities* in the environment. For the book-keeping of which objects of the environment have been told which identities, a well-typed component must take into account the *relation* of object names from the assumption context Δ amongst each other, and the knowledge of objects from Δ about those exported by the component, i.e., those from Θ. [6] In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component:

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

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$. We write $o_1 \hookrightarrow o_2$ ("o_1 may know o_2") for pairs from these relations. As mentioned, the component does not have full information about the complete system and thus it must make worst-case assumptions concerning the proliferation of knowledge. These worst-case assumptions are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of objects *from* Δ the component maintains. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

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

Note that we close $\hookrightarrow$ only wrt. environment objects, but not wrt. objects at the *interface,* i.e., the part of $\hookrightarrow \subseteq \ \Delta \times \Theta$. We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\!\hookrightarrow$ for that union. As judgment, we use $\Delta; E_\Delta \vdash v_1 \leftrightharpoons v_2 : \Theta$, respectively $\Delta; E_\Delta \vdash v_1 \leftrightharpoons\!\!\hookrightarrow v_2 : \Theta$. For Θ, E_Θ, and Δ, the definitions are applied dually.

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

2.2 External steps

The external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments (cf. Table 3). Besides internal steps a component exchanges information

[6] Besides the relationships amongst objects, we need to keep one piece of information concerning the "connectivity" of *threads*. To exclude situations where a known thread leaves the component into one clique of objects but later returns to the component coming from a different clique without connection to the first, we remember for each thread that has left the component the object from Δ it has left into.

with the environment via *calls* and *returns.* Using a lazy instantiation scheme for cross-border object creation, there are no separate external labels for *new*-steps. Thus, core labels γ are of the form $n\langle \mathit{call}\ o.l(\vec{v})\rangle$ and $n\langle \mathit{return}(v)\rangle$. Names may occur bound in a label $\nu(n{:}T).\gamma$, and receiving and sending labels are written as $\gamma?$ and $\gamma!$. In this extended abstract, we omit the typing premises in the operational rules ("only values consistent with the static typing assumptions may be received" and the like) as they are straightforward and we concentrate on the novel aspects, namely the connectivity information.

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

Table 2. Labels

Connectivity assumptions and commitments As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ overapproximates the actual connectivity of the component, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. *Incoming* calls update the commitment context E_Θ in that it remembers that the callee o_2 now knows (or rather may know) the arguments $\vec{v}$, and furthermore that the thread n has entered o_2. For incoming communication (cf. rules CALLI$_2$ and RETI)[7] we require that the sender be acquainted with the transmitted arguments.

For the role of the caller identity o_1, a few more words are in order. The antecedent of the call-rules requires, that the caller o_1 is acquainted with the callee o_2 and with all of the arguments. However, the caller is *not* transmitted in the label which means that it remains anonymous to the callee.[8] To gauge, whether an incoming call is possible and to adjust the book-keeping about the connectivity appropriately, in particular when returning later, the transition chooses among possible sources of the call. With the sole exception of the initial (external) step, the scope of at least *one* object of the calling clique must have escaped to the component, for otherwise there would be now way of the caller to address o_2 as callee. In other words, for at least one object o_1 from the clique of the actual caller (which remains anonymous), the judgment $\Delta \vdash o_1{:}c$ holds prior to the call. Furthermore it must be checked that the incoming thread originates

[7] We omit rules dealing with the initial situation where the first thread crosses the interface between environment and component.

[8] Of course, the caller may transmit its identity to the callee as part of the arguments, but this does not reveal to the callee who "actually" called. Indeed, the actual identity of the caller is not needed; it suffices to know the *clique* of the caller. As representative for the clique, an equivalence class of object identities, we simply pick one object.

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; n \hookrightarrow o_2 \hookrightarrow \vec{v}) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta} \quad t_{blocked} = let\ x{:}T' = o_2'\ blocks\ for\ o_1'\ in\ t \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel n\langle t_{blocked}\rangle : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ return\ to\ o_1\ x; t_{blocked}\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CalliI}_2$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle return(v)\rangle! \qquad (\Theta', \Delta') = fn(v) \cap \Phi \qquad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; (n \hookrightarrow o_1 \hookrightarrow v) \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(\acute{C}, \Theta') \setminus n \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = o_2\ return\ to\ o_1\ v\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{RetO}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle call\ o_2.l(\vec{v})\rangle! \quad (\Theta', \Delta') = fn(n\langle call\ o_2.l(\vec{v})\rangle \cap \Phi \\ \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \quad o_2 \in dom(\acute{\Delta}) \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; (n \hookrightarrow o_2 \hookrightarrow \vec{v}) \qquad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(\acute{C}, \Theta') \setminus n \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\ x{:}T = [o_1]\ o_2.l(\vec{v})\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CallO}$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle return(v)\rangle? \qquad dom(\Delta', \Theta') \subseteq fn(v) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta', (n \hookrightarrow o_1 \hookrightarrow v) \qquad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta', (o_2 \hookrightarrow (\Delta', \Theta')) \setminus n \\ \acute{\Delta}; \acute{E}_\Delta \vdash o_2 \leftrightharpoons\!\hookrightarrow v : \acute{\Theta} \end{array}}{\Delta; E_\Delta \vdash C \parallel n\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel n\langle t[v/x]\rangle : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{RetI}$$

$$\frac{c \in dom(\Delta)}{\Delta; E_\Delta \vdash n\langle let\ x{:}c = new\ c\ in\ t\rangle : \Theta; E_\Theta \rightsquigarrow \Delta; E_\Delta \vdash \nu(o_3{:}c).n\langle let\ x{:}c = o_3\ in\ t\rangle : \Theta; E_\Theta}\ \textsc{NewO}_{lazy}$$

Table 3. External steps

from a group of objects in connection with the one to which the thread had left the component the last time: $\acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 : \acute{\Theta}$. Once chosen, the assumed identity of the caller is remembered as part of the return-syntax.

It is worth mentioning that in rule RetI the proviso that the callee o_2 knows indirectly the caller o_1, i.e., $\Delta; E_\Delta \vdash o_2 \leftrightharpoons\!\hookrightarrow o_1 : \Theta$ is not needed. Neither is it necessary to require in analogy to the situation for the incoming call that the thread is acquainted with the callee. If fact, both requirements will be automatically assured for traces where calls and returns occur in correct manner.

A commonality for incoming communications from a thread n is that the (only) pair $n \hookrightarrow o$ for some object reference o is removed from E_Δ, for which we write $E_\Delta \setminus n$. While E_Δ imposes restrictions for incoming communication, the commitment context E_Θ is *updated* when receiving new information. For instance in $\textsc{CalliI}_2$, the commitment $\acute{E}_\Theta$ after reception marks that now the callee o_2 is acquainted with the received arguments and furthermore that the thread n is visiting (for the time being) the callee o_2. For *outgoing* communication, the E_Δ

and E_Θ play dual roles. In the respective rules, $E(\acute{C}, \Theta')$ stands for the actual connectivity of the component after the step, which needs to be made public in the commitment context, in case new names escape to the environment.

Scoping and lazy instantiation In the explanation so far, we omitted the handling of bound names, in particular bound object references. In the presence of classes, a possible interaction between component and environment is instantiation. Without constructor methods and assuming an infinite heap space, instantiation itself has no immediate, observable side-effect. An observable effect is seen only at the point when the object is accessed.

Rule NEWO$_{lazy}$ 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.

For incoming calls, for instance, the binding part is of the form (Δ', Θ') where we mean by convention, that Δ' are the names being added to Δ, and analogously for Θ' and Θ. For object names, the distinction is based on the class types. For thread names, the reference is contained in Δ' and Θ', and class names are never transmitted. For the object names in the incoming communication Δ' contains the external references which are freshly introduced to the component by scope extrusion. Θ' on the other hand are the objects which are *lazily instantiated* as side-effect of this step, and which are from then on part of the component. In the rules, the newly instantiated objects are denoted as $C(\Theta')$.

Note that whereas the acquaintance of the caller with the arguments transmitted free is checked against the current assumption, acquaintance with the ones transmitted bound is added to the assumption context.

3 Trace semantics and ordering on traces

Next we present the semantics for well-typed components, which, as in the object-based setting, takes the sequences of external steps of the program fragment as starting point.

Not surprisingly, a major complication now concerns the connectivity of objects. In this context, the caller identity, while not visible by the callee, plays a crucial role in keeping track of assumed connectivity, in particular to connect the effect of a return to a possible caller clique. To this end, the operational semantics hypothesizes about the originator of incoming calls and remembers the guess as "auxiliary" annotation in the code for return (cf. rule L-CALLI$_2$ from Table 3).

The (hypothetical) connectivity of the environment influences what is observable. Very abstractly, the fact the observer falls into a number of independent cliques increases the "uncertainty of observation". We can point to two reasons

responsible for this effect. One is that separate observer cliques cannot determine the relative order of events concerning only one of the environment cliques. To put it differently: a clique of objects can only observe the order of events *projected* to its own members. We will worry about this later when describing the all possible reorderings or interleavings of a given trace. Secondly, separate observers cannot cooperate to *compare identities.* This means, as long as separated, the observers cannot find out whether identities sent to each of them separately are the same or not. In terms of projections to the observing clique it means that local projections are considered up to α-conversion, only.

The above discussion should not mislead us to think that the behavior of two observing cliques is completely independent. One thing to keep in mind is that the observers can merge. This means that identities, separate and local prior to the merge, become comparable, and the now joint clique can find out whether local interaction of the past used the same identities or not. The absolute order of local events of the past, however, cannot be reconstructed after merging.

Another more subtle point, independent from merging of observers, is that to a certain degree, the events local to one clique *do influence* interaction concerning another clique. This in other words implies that considering *only* the separate local projections of a global behavior to the observers is too abstract to be sound.

To understand the point, consider as informal example a situation of a component C_1 with two observing cliques in the environment and a sequence s of labels at the interface of the component being observed. Assume further that s_1 is the projection of s to the first observer and s_2 the projection to the second, and assume that $s = s_1 s_2$ meaning that s_1 precedes s_2 when considered as global behavior. For sake of the argument, assume additionally that C_1 is not able to perform the interaction in the swapped order $s_2 s_1$. Given a second component C_2 being more often successful, i.e., that $C_1 \sqsubseteq_{may} C_2$, what does this imply for C_2's behavior? The definition of may-preorder is given in Section 4. For the moment, being successful can be thought of being able to reach some predefined point which counted as success.

Since the environment can be programmed in such a way that it reports success only after completing s_1 resp. s_2, it is intuitively clear that C_2 must be able to exhibit s_1 resp. s_2. But the environment cannot observe whether C_2 performs s_1 and s_2 *in the same run,* as does C_1. We can only be sure that there is a run of C_2 which is able to do s_1 and a (potentially different) one which does s_2, each of which is taken as independent sign of success. This does not mean, however, that the order of $s_1 s_2$ does not play a role at all. Consider for illustration the situation where C_2 can perform $s_2 s_1$ but not $s_1 s_2$ as C_1: In this case, $C_1 \not\sqsubseteq_{may} C_2$, i.e., C_2 is not successful while C_1 is, namely in an environment where s_2 is possible and reports success but s_1 *can be hindered from completion.* In other words, taking the behavior $s_1 s_2$ of C_1 as starting point we cannot consider in isolation the fact that s_2 is possible by C_2 as well, the order of s_1 preceding s_2 is important inasmuch as it s_1 can *prevent* success for s_2. So $C_1 \not\sqsubseteq_{may} C_2$ and the fact that C_1 performs the sequence $s_1 s_2$ means, that C_2 can perform s_2 after a prefix of s_1. Since the common environment has already

proven in cooperation with C_1 that it is able to perform s_1, it cannot prevent success of C_2 by blocking.

To sum up and independently of the details: to capture the observable behavior appropriately, we need to be able to define the projection of the external steps to the observer cliques. Now the labels for method calls in the external semantics do not contain information concerning the caller, which means given a trace as a sequence of labels, we have no indication for incoming calls concerning the originating environment clique.[9]

A way to remedy this lack of information is to augment the labels as recorded in the traces by the missing information. So instead of the call label described in Section 2.2, we use $n\langle[o_1]\mathit{call}\ o_2.l(\vec{v})\rangle$ as annotated call label, where o_1 denotes the caller, respectively the clique of the caller. The augmented transitions are generated simply by using the caller rules from Table 3 where the caller is added to the transition labels in the obvious way.

A trace of a well-typed component is a sequence s of external steps, where we write $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{s}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$. For $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{\epsilon}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$, we write shorter $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \Longrightarrow \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$.

With this information we can define the *projection* of a trace onto a clique as the part of the sequence containing all the labels with objects from that clique. Remember that a clique of an object $o \in \Theta$ consists of all objects from Θ acquainted with o. Thus the equivalence $\leftrightharpoons$ partitions Θ into equivalence classes, and formally we could write $[o]_{/_{E_\Theta}}$ or $[o]_{/_{\leftrightharpoons}}$ for that equivalence class. For simplicity, we often just write $[o]$.

The definition of projection of an (augmented) trace onto a clique of environment objects is straightforward, one simply jettisons all actions not belonging to that clique. One only has to be careful dealing with exchange of bound names, i.e., scope extrusion, since names sent for the first time to a clique are to be considered as *locally fresh,* even if the name may globally be known to other environment cliques.

We can now define the order on traces as follows.

Definition 1. $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$*, if the following holds. If* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \stackrel{sa}{\Longrightarrow} \Delta'; E'_\Delta \vdash C'_1 : \Theta'; E'_\Theta$*, then* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta \stackrel{t}{\Longrightarrow} \Delta''; E''_\Delta \vdash C''_2 : \Theta''; E''_\Theta$ *such that*

- $t \downarrow_{[o'']} = sa \downarrow_{[o_a]}$ *for some clique* $[o'']$ *according to* $\Theta''; E''_\Theta$ *and when* $[o_a]$ *is the environment clique to which label a belongs, and*
- *for all cliques* $[o'']$ *according to* $\Delta''; E''_\Delta$*, there exists a clique* $[o']$ *according to* $\Delta'; E'_\Delta$ *such that* $t \downarrow_{[o'']} \preccurlyeq sa \downarrow_{[o']}$.

[9] For outgoing calls, the relevant environment clique is mentioned explicitly as the receiver of the call. Concerning returns, the concerned environment clique is determined by the matching call.

4 Notion of observation

Full abstraction is a comparison between two semantics, where the reference semantics to start from is traditionally *contextually* defined and based on a some notion of *observability.*

As starting point we choose, as [7], a (standard) notion of semantic equivalence or rather semantic implication —one program allows at least the observations of the other— based on a particular, simple form of contextual observation: being put into a context, the component, together with the context, is able to *reach* a defined point, which is counted as the successful observation. A context $\mathcal{C}[\_]$ is a program "with a hole". In our setting, the hole is filled with a program fragment consisting of a *component* C in the syntactical sense, i.e., consisting of the parallel composition of (named) classes, named objects, and named threads, and the context is the rest of the programs such that $\mathcal{C}[C]$ gives a well-typed *closed* program $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$, where closed means that it can be typed in the empty contexts, i.e., $\vdash C' : ()$.

To report success, we assume an external class with a particular success-reporting method. So assume a class c_b of type $[\![succ : () \to none]\!]$, abbreviated as barb. A component C *strongly barbs on* c_b, written $C \downarrow_{c_b}$, if $C \equiv \nu(\vec{n}{:}\vec{T}, b{:}c_b).C' \parallel n\langle let\, x{:}none = b.succ()\, in\, t\rangle$, i.e., the call to the success-method of an instance of c_b is enabled. Furthermore, C *barbs on* c_b, written $C \Downarrow_{c_b}$, if it can reach a point which strongly barbs on c_b, i.e., $C \Longrightarrow C' \downarrow_{c_b}$. We can now define may testing preorder [6] as in [7].

Definition 2 (May testing). *Assume* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta$ *and* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$. *Then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$, *if* $(C_1 \parallel C) \Downarrow_{c_b}$ *implies* $(C_2 \parallel C) \Downarrow_{c_b}$ *for all* $\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C : \Delta; E_\Delta$.

5 Full abstraction

The proof that may-testing coincides with order on traces given in Definition 1 has two directions: compared to $\sqsubseteq_{may}$, the relation $\sqsubseteq_{trace}$ is neither too abstract (soundness) nor too concrete (completeness).

For lack of space, we simply state the soundness result here. The proof is rather similar to the one for the object-based case [7] and rests on the ability to compose a component and an environment, performing complementary traces, into one global program (plus the dual property of decomposition). We refer to the full version [3] for details.

Proposition 1 (Soundness). *If* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, *then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$.

Completeness asserts the reverse direction:

Proposition 2 (Completeness). *If* $\Delta; E_\Delta \models C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$, *then* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$.

Concerning completeness, we sketch here one core aspect part of the argument. At the heart, completeness is a constructive argument: given a trace s, construct a component C_s that exhibits the trace s and not simply realize the trace, but realize it *exactly,* up-to unavoidable reordering and prefixing.

Legal traces To do so, we must first characterize which traces (the "legal" ones) can occur at all, and again the crucial difference to the object-based case is to take connectivity into account to exclude impossible combinations of transmitted object names and threads.

The legal traces are specified by a system for judgments of the form $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\,\Theta; E_\Theta$ stipulating that under the type and relational assumptions Δ and E_Δ and with the commitments Θ and E_Θ, the trace s is legal. Three exemplary rules for legal traces are shown in Table 4; not shown are two dual rules for outgoing calls and incoming returns, and furthermore two rules specifying the situation for the initial calls, which are similar to L-CallI. For simplicity, we omit premises dealing with static aspects of typing, as we did for the external semantics. As in the operational semantics, the caller identity, even if not part of the label, is guessed and remembered, here in the history r. The premise $\Delta \vdash r \rhd a : \Theta$ asserts that after r, the action a is enabled, and $\mathit{pop}\ n\ r$ picks the call matching the return in question. See [3] for details.

$$\frac{}{\Delta; E_\Delta \vdash r \rhd \epsilon : \mathit{trace}\,\Theta; E_\Theta}\ \text{L-Empty}$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle \mathit{call}\ o_2.l(\vec{v})\rangle? \qquad \Delta \vdash o_1 : c_1 \qquad \Delta \vdash r \rhd a : \Theta \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; n \hookrightarrow o_2 \hookrightarrow \vec{v}) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a_{o_1} \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\,\Theta; E_\Theta}\ \text{L-CallI}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle \mathit{return}(v)\rangle! \qquad \mathit{pop}\ n\ r = \nu(\Delta'', \Theta'').\ n\langle [o_1]\mathit{call}\ o_2.l(\vec{v})\rangle? \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; n \hookrightarrow o_1 \hookrightarrow v \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow (\Theta', \Delta') \setminus n \\ \acute{\Theta}; \acute{E}_\Theta \vdash o_2 \leftrightharpoons\!\hookrightarrow v : \acute{\Delta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\,\Theta; E_\Theta}\ \text{L-RetO}$$

Table 4. Legal traces

6 Conclusion

Inspired by the work of [7], we presented an operational semantics of a class-based, object-oriented calculus with multithreading. The seemingly innocent step from an *object-based* setting as in [7] to a framework with classes requires quite some extension in the operational semantics to characterize the possible behavior of a component. In particular it is necessary to keep track of the potential

connectivity of objects of the environment to exclude impossible communication labels. It is therefore instructive, to review the differences in this conclusion, especially to try to understand how the calculus of [7] can be understood as a special case of the framework explored here.

The fundamental dichotomy underlying the observational definition of equivalence is the one between the inside and the outside: program or component vs. environment or observer, or in game-theoretical terms, player vs. opponent. This leads to the crucial difference between object-based languages, instantiating from objects, and class-based language, instantiating from classes: In the class-based setting, instantiation may *cross the demarcation line between component and environment,* while in the object-based setting, this is not possible: the program only instantiates program objects, and the environment only objects belonging to the environment. All other complications follow from this difference, the most visible being that it is necessary to represent the dynamic object structure into the semantics, or rather an approximation of the connectivity of the environment objects. Another way to see it is, that in the setting of [7], there is only *one clique* in the environment, i.e., in the worst case, which is the relevant one, all environment objects are connected with each other. Since the component cannot create environment objects (or vice versa), new isolated cliques are never created. The object-based case can therefore be understood by invariantly (and trivially) taking $E_\Delta = \Delta \times (\Delta + \Theta)$, while in our setting, E_Δ may be more specific.

Further related work [12] investigates the full abstraction problem in an object calculus with subtyping. The setting is a bit different from the one as used here as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. [4] presents a full abstraction result for the π-calculus, the standard process algebra for name passing and dynamically changing process structures. The extensional semantics is given as a domain-theoretic, categorical model, and using bisimulation equivalence as starting point, not may testing resp. traces as here. [13] gives equational full abstraction for standard translation of the polyadic π-calculus into the monadic one. Without additional information, the translation is not fully abstract, and [13] introduces graph-types as an extension to the π-calculus sorting to achieve full abstraction. The graph types abstract the *dynamic* behavior of processes. In capturing the dynamic behavior of interaction, Yoshida's graph types are rather different from the graph abstracting the connectivity of objects presented here. Recently, Jeffrey and Rathke [8] extended their work on trace-based semantics from an object-based setting to a core of *Java* (called *Java Jr.*), including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*.

Acknowledgements We thank Andreas Grüner for careful reading, discussing, helping to clarify and improving a number of half-baken previous versions of the document. Likewise Karsten Stahl and Harald Fecher for "active listening" even

to the more Byzantine details and dead ends of all this. We are also indepted to Ben Lukoschus for helping with some of the more arcane TEX-stunts and to Willem-Paul de Roever for spotting a number of sloppy points. Finally, we thank the reviewers for their insightful remarks.

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. Preliminary technical report, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Jan. 2005.
4. M. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the π-calculus (extended abstract). In *Proceedings of LICS '96*, pages 43–54. IEEE, Computer Society Press, July 1996.
5. 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.
6. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
7. 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.
8. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. 2005. Submitted for publication.
9. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
10. A. M. Pitts and D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In A. M. Borzyszkowski and S. Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Sept. 1993.
11. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
12. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.
13. N. Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.