INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

A structural operational semantics for a concurrent class calculus

Erika Ábrahám   Marcello M. Bonsangue

Frank S. de Boer   Martin Steffen

PAX OPTIMA RERUM

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

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

A structural operational semantics for a concurrent class calculus

Erika Ábrahám   Marcello M. Bonsangue

Frank S. de Boer   Martin Steffen

e-mail: eab@informatik.uni-freiburg.de, marcello@liacs.nl,
F.S.de.Boer@cwi.nl, ms@informatik.uni-kiel.de

A Structural Operational Semantics for a Concurrent Class Calculus

August 17, 2003

Erika Ábrahám[1,2], 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 ν-calculus has been investigated as a core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects. From an abstract point of view, 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 technical report extends the concurrent ν-calculus by introducing classes and explores some of the semantical consequences. The semantics will serve as the basis for a proof of full abstraction wrt. to a may-testing based notion of observability.

Keywords: multithreading, class-based object-oriented languages, formal semantics.

Table of Contents

1 Introduction

The semantics of a program is what "it means". Alas,[5] not only are there many
possibilities to describe "the" meaning of a program —denotational, operational,
logically, equationally with various intermediate shades— but also there are
many possible choices what the meaning should actually be.

A standard approach, which is also philosophically appealing, is not to start
defining what program *is* but what one (wishes to) *observe* about the program or
more generally the program fragment. Still, there are of course different choices,
but often the choice of a notion of observability is much simper to define, to
understand, and to defend, in one word, less arbitrary, than an outright definition
of the semantics. The coincidence of a (in most cases) denotational notion of
equivalence and observable equivalence is known as the *full abstraction* problem
and has been studied for numerous languages and settings.

Our particular interest in the field are semantics of multithreaded, object-
oriented languages in the style of *Java*. In the context of concurrent, *object-
based* programs, Jeffrey and Rathke [3] offered an answer to the full abstraction
problem. Starting from may-testing as a very simple notion of observation, their
result roughly says 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
of the components. At this level, the result is unspectacular, since indeed it is
intuitively clear that in the chosen setting, the only possible way to observe

---

[5] or fortunately, depending on the standpoint ...

something about a set of objects and threads is to exchange messages. It should be equally clear, however, that for the language featuring multithreading, object references with aliasing, and creation of new objects and threads, the details of defining the semantics and proving the full abstraction result are all but trivial.

The result in [3] is developed within the concurrent ν-calculus [2], which has been proposed as a core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects. It is an extension of the sequential ν-calculus [5] and stands in the tradition of various object calculi (cf. [1] for one definite reference) and also the π-calculus [4, 6].

One distinctive feature of the ν-calculus is that it is *object-based,* which in particular means that there are no *classes* as templates for new objects.[6] This is in contrast to the mainstream of object-oriented languages whose code is organized in classes. This report addresses therefore the following question:

> What changes when switching from an object-based to a class-based setting?

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 the methods, and thread, 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.

May it as it be, what is important in our context is that now the division between the component or program fragment under observation, and its environment also separates *classes:* There are classes internal to the component and those belonging to the environment which plays the role of the program's observer. 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 has as a consequence that the component creates only component-objects and dually the environment only environment objects.

To understand the bearing on the semantics of this change, we must be aware that the interesting part of the problem is not so much to just cover the possible behavior at the interface —there is little doubt that sequences of calls, returns, and instantiations with enough information at the labels would do— but to characterize it *exactly,* i.e., to exclude impossible environment interaction. As

---

[6] The terms "object-based" and "object-oriented" are sometimes used to distinguish between two flavors of languages with objects: object-oriented languages, in this manner of speaking, support classes and inheritance, whereas object-based languages do without classes. Instead, they offer more complex operations on objects, for instance general method update.

an obvious example, a trace with two consecutive calls from the same thread without a return in between should not be part of the behavior of a component.

Let us concentrate on the issue of instantiation across the demarcation line between component and its environment, and imagine that the component creates an instance of an environment class. The first question is: does this yield a component object or an environment object? As the code of the object is provided by the external class which is in the hand of the observer, the interaction between the component and the newly created object can lead to observable effects and must therefore be traced. In other words, instances of environment classes belong to the environment, and symmetrically those of internal classes to the component.

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 o_3.
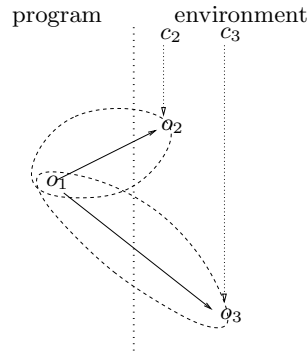

Fig. 1. Two instances of external classes

Now in this situation it is impossible, that there be an incoming call from the environment carrying both names o_2 and o_3, since the only entity which is aware of both references is o_1. Unless the component gives away the reference to the environment, o_2 and o_3 are completely separated.

Thus, in order to exclude impossible combinations of object reference in the communication labels, the component has to keep track 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 over-approximate 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 include 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.

To summarize: The report investigates a class-based variant of the concurrent ν-calculus, formalizing the ideas sketched above about cliques of objects. Instantiation itself, even across the environment-program boundary, is not observable, since the calculus does not have constructor methods. Therefore we present also a variant of the semantics with *"lazy instantiation"*, i.e., where an externally created object is created only at the point, when it is actually accessed the for the first time.

Overview The report is organized as follows. We start in Section 2 and Section 3 with the syntax of the calculus and its type system. In this part, the development is rather close to the one in [3]. Section 4 contains the operational semantics, including a variant with lazy instantiation. Finally, Section 5 contains remarks about related and future work. Appendix A presents and alternative formulation of semantics which explicitly reprents connectivity information as part of the component itself, which simplifies the static checking.

2   A concurrent class calculus

This section presents the syntax of the class-based calculus we will use for our study. Indeed, it is more or less a syntactic extension of the concurrent object calculus or concurrent ν-calculus from [2, 3].

Compared to the object-based concurrent ν-calculus, the basic change is the introduction of *classes,* where a class is a named collection of methods just as an object in the object calculus. One difference between an object and a class concerns the nature of its name or identifier. Class names are the literals introduced when defining the class; the 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 *dynamcic,* and especially they can be created freshly by instantiating a class. There are no constant object names; the only way to get a new reference is instatiation.[7]

---

[7] The calculus does not contain an explicit constant name for the undefined reference, e.g. *nil*.

The calculus is a *typed* language; also the operational semantics will be developed for well-typed program fragments, only. Besides base types B if wished —we will allow ourselves integers, booleans, . . . , where convenient— the type *none* represents the absence of a return value and *thread* is the type for a named thread. The type $[\![l_1{:}U_1, \dots , l_k{:}U_k]\!]$ of a class fixes the method labels l_1 to l_k and the method types, where each method is typed by the functional type from the tuple of inputs to the return type. The name n of a class serves as the type for the instances of the class. The grammar is shown in Table 1.

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

Table 1. Types

A program is given by a collection of classes, where the empty collection is denoted by **0**. A class $n[\![O]\!]$ carries a name n and defines the implementation of its methods, and analogously for objects. A method $\varsigma(n{:}T).\lambda(x_1{:}T_1, \dots , x_n{:}T_k).t$ provides the definition of the method body abstracted over the formal parameters of the method. The name parameter n plays a specific role: It is the "self" parameter wich is bound to the identity of the object upon method call. The body itself is a sequential piece of code, i.e., an (anonymous) *thread.* 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"), but directly created by providing the code. A thread is either a value v, or a sequence of expressions, where the *let*-construct is used for local declarations and sequencing; *stop* stands for the deadlocked or terminated thread. Besides threads, expressions comprise conditionals and method calls, furthermore object creation via instantiation, creation of new threads, and a reference to the current thread. Values, finally, are either variables x or names n (and *true, false*, $0, 1, \dots$ when convenient). For the names, 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. The abstract syntax is displayed in Table 2.

We further will use the following syntactic abbreviations. The sequential composition $t_1; t_2$ of two threads stands for *let* $x{:}T = t_1$ *in* t_2, where x does not occur free in t_2. Instance variables or fields are treated as specific form of methods, namely of empty parameter list, i.e., an instance variable declaration $f = v$ is expanded to $\varsigma n{:}T.\lambda().v$, a field access $x.f$ to $x.f()$, and field update to $x := v$ to $v.f \Leftarrow \varsigma o{:}T.\lambda().v'$.

$$
\begin{array}{lll}
C ::= \mathbf{0} \mid R \parallel R \mid \nu(n{:}T).R \mid n[\![O]\!] \mid n[O] \mid n\langle t\rangle & \text{program} \\
O ::= l = m, \dots, l = m & \text{object} \\
m ::= \varsigma(n{:}T).\lambda(x{:}T, \dots, x{:}T).t & \text{method} \\
t ::= v \mid stop \mid \mathit{let}\, x{:}T = e\, \mathit{in}\, t & \text{thread} \\
e ::= t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e & \text{expr.} \\
\quad\mid\quad v.l(v, \dots, v) \mid n.l \Leftarrow m \mid \mathit{currentthread} \\
\quad\mid\quad \mathit{new}\, n \mid \mathit{new}\langle t\rangle \\
v ::= x \mid n & \text{values}
\end{array}
$$

Table 2. Abstract syntax

In the class-based setting, furthermore, we will not make use of general method update, and we additionally disallow (read and write) references to fields across object boundaries.[8]

3   Type system

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

Table 3, to begin with, defines the typing on the level of global configurations, i.e., on "sets" of threads, objects, and classes, all named. On this level, the typing judgments are of the form $\Delta \vdash C : \Theta$, where Δ and Θ are finite mappings from names to types. In the judgment, Δ plays the role of the typing *assumptions* about the environment, and Θ the *commitments* of the configuration, i.e., the names offered to the environment. This means, Δ must contain *at least* all external names referenced by C and dually Θ mentions *at most* the names offered by C.

The empty configuration is denoted by $\mathbf{0}$; it is well-typed in any context and exports no names (cf. rule T-EMPTY). Two configurations in parallel can refer mutually to each other's commitments, and together offer the union of their names (cf. rule T-PAR). It will be an invariant of the operational semantics that the identities of parallel entities (except for thread names) are unique. Therefore, Θ_1 and Θ_2 in the rule for parallel composition are merged disjointly, as far as the object and class references are concerned.

On the static level of the type system, the ν-binder hides the bound name within its scope (cf. rule T-NU). Note that for $\nu(n{:}T).C$, the ν-bound name is *not* added to the assumption context Δ, but to the commitment Θ. This means, the ν-construct not only introduces a local scope for its bound name but

---

[8] [3] are slightly more general in this respect, they only disallow write-access — including method update— across component boundaries, by introducing the semantic notion of write closedness. The theory does not depend on this difference. Therefore we content ourselves here with the simpler syntactic restriction which completely disallows field access across object boundaries.

asserts something quite stonger, namely the *existence* of a likewise named entity. This highlights one difference of let-bindings for variables and the introduction of names via the ν-operator: the language construct to introduce names is the *new*-operator, which opens a new local scope and a named component running in parallel. The let-bound variable is *stack* allocated and thus checked in a stack-organized variable context Γ. Names created by *new* are *heap* allocated and thus checked in a "parallel" context (cf. the assumption-commitment rule T-PAR). The instantiated object will be available in the exported context Θ by rule T-NOBJ. The rules for the named entities introduce the name and its type into the commitment (cf. rules T-NOBJ, T-NCLASS, T-NTHREAD).

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

$$\frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta} \ \text{T-NU}$$

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

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

Table 3. Static semantics (configurations)

The typing rules of Table 4 formalize typing judgements for threads and objects and their syntactic sub-constituents. Besides assumptions about the provided names of the environment kept in Δ as before, the typing is done relative to assumption about occuring free variables. They are kept separately in a variable context Γ, a finite mapping from variables to types.

The typing rules are rather straightforward and in many cases identical to the ones for the concurrent ν-calculus [3]. Different from the object-based setting are the ones dealing with classes. Rule T-CLASS is the introduction rule for class types, the rule of instantiation of a class T-NEWC requires reference to a class-typed name. Note also that the deadlocking expression *stop* has every type.

4 Operational semantics

Next we present the *operational* semantics of the calculus. Again, the formalization is quite close to the one for the object calculus, except the parts dealing with classes. The basic steps of the semantics are given in two levels: internal

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

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

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

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![\ldots, l{:}T_1 \times \ldots \times T_k \to T, \ldots]\!] \quad \Gamma;\Delta \vdash v_1 : T_1 \ldots \Gamma;\Delta \vdash v_k : T_k}{\Gamma;\Delta \vdash v.l(v_1, \ldots, v_k) : T} \text{ T-Call}$$

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

$$\frac{\Gamma;\Delta \vdash c : [\![T]\!]}{\Gamma;\Delta \vdash new\ c : c} \text{ T-NewC} \qquad \frac{\Gamma;\Delta \vdash t : T}{\Gamma;\Delta \vdash new\langle t\rangle : thread} \text{ T-NewT}$$

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

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

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

$$\frac{}{\Gamma;\Delta \vdash stop : T} \text{ T-Stop}$$

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

$$\frac{}{\Gamma;\Delta \vdash block : T} \text{ T-Block} \qquad \frac{\Gamma;\Delta \vdash v : T}{\Gamma;\Delta \vdash return[o_1]\ v : T'} \text{ T-Return}$$

Table 4. Static semantics (2)

steps, i.e., those whose effect is completely confined within a configuration, and those with external effect.

4.1   Internal steps

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

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

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

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

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

$c[\![O]\!] \parallel n\langle let\ x{:}c = new\ c\ in\ t\rangle \rightsquigarrow c[\![O]\!] \parallel \nu(o{:}c).(o[O] \parallel n\langle let\ x{:}c = o\ in\ t\rangle)$ NEWO$_i$

$n\langle let\ x{:}T = new\langle t\rangle\ in\ t_1\rangle \rightsquigarrow \nu(n_2{:}T).(n\langle let\ x{:}T = n_2\ in\ t\rangle \parallel n_2\langle t\rangle)$ NEWT

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

$o[O] \parallel n\langle let\ x{:}T = o.l(\vec{v})\ in\ t\rangle \xrightarrow{\tau} o[O] \parallel n\langle let\ x{:}T = O.l(o)(\vec{v})\ in\ t\rangle$ CALL$_i$

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

Table 5. Internal steps

We start in Table 5 with the internals steps, where we distinguish between *confluent* steps, written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$, i.e., those potentially leading to race conditions in the context of threads running in parallel. For instance, the first 5 rules of the table deal with the basic sequential constructs, all as $\rightsquigarrow$-steps. The basic evaluation mechanism is substitution (rule RED). Note that the rule requires that the leading let-bound variable of a thread can be replaced only by *values,* which makes the reduction strategy deterministic, at least per thread. The *stop*-thread terminates for good, i.e., the rest of the thread will never be executed (cf. rule STOP).

The step NEWO$_i$ describes the creation of an instance of an *internal* class $c[\![O]\!]$, i.e., a class whose name is contained in the configuration. Note that instantiation is a confluent step. The object in O is taken as template for the created object. The identity of the object is new and local —for the time being— to the instantiating thread; the new named object and the thread are thus enclosed in a ν-binding. Rule CALL$_i$ treats an internal method call, i.e., a call to an object contained in the configuration. In the step, $O.l(o)(\vec{v})$ stands for $t[o/n][\vec{v}/\vec{x}]$, when the object O equals $[\ldots, l = \varsigma(s{:}T).\ \lambda(\vec{x}{:}\vec{T}).t, \ldots]$. Note also that the step is a $\xrightarrow{\tau}$-step, not a confluent one. The same holds for field update in rule FUPDATE, where $[l_1 = m_1, \ldots l_k = m_k, f = v'].f := v$ stands for $[l_1 = m_1, \ldots, l_k = m_k, f = v, \ldots]$. Note further, that instances of a component

class invariantly belong to the component and not the environment. This means that an instance of a component class resides after instantiation in the component, and named objects will never be exported from the component to the environment or vice versa; of course, *references* to objects may well be exported.

The reduction relations from above are used modulo *structural congruence,* which captures the algebraic properties of parallel composition and the hiding operator. The basic axioms for $\equiv$ are shown in Table 6 and the relation is imported into the reduction relations in Table 7. Note that besides the structural rules, all syntactic entities are always tacitly understood modulo α-conversion.

$\mathbf{0} \parallel C \equiv C \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$

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

Table 6. Structural congruence

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

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

Table 7. Reduction modulo congruence

4.2   External behavior of a component

The *external* behavior of a component is given in terms of labeled transitions. The transitions describe 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.

For single labeled steps, one has to insist, for instance, that calling a method of an external object refers to an object actually present in the environment, or dually that incoming calls have as target only objects exported to the outside, and furthermore that the communicated values are in accordance to the well-typedness assumption. Therefore, at least in first approximation, the transitions

are given between typing judgments $\Delta \vdash P : \Theta$. Again this general starting point is similar to the situation for the object calculus in [3].

A further local condition concerns which *combinations* of names can occur in communications. This phenomenon does not occur in the object based setting and merits a closer discussion before we embark on the formalization in the following section.

To take a simple example, assume the component creates an instance of a class resident in the environment. Similar to the internal steps as given in Table 5, this will be done by some thread of the component executing a *new-*statement, with the difference that the instantiated class does not occur inside the component as in rule NEWO_i, but is listed in the assumption context Δ.

As the class is part of the environment and thus in the hand of the observer, it can be used to make observations via its instances. Consequently, its instance belongs to the environment, as well, and communication from and to this object will be traced. While occurring likewise at the interface between the component and the environment, however, the *instantiation itself* cannot be used by the context to make any observations about the component. This is a consequence of two facts. First, our language does not support *constructors* which, in the hand of the environment, could be used to make distinguishing observations. Secondly, exchanging a class by another and thus exchanging its instances does not make a difference in the overall behavior *unless* the component communicates with the instances; the pure existence of one object or another does not make any difference.[9]

Assume now that the component creates *two* instances of an external class or of two different external classes; the class types of the two objects do not play a role. As just explained, the objects named o_1 and o_2, say, are themselves part of the environment. Is it possible in this situation that a communication occurs where o_1 issues a call to an object of the component with o_2 as argument? Clearly the answer is no, unless the component has *given away* the identity of o_2 to o_1, since otherwise there is no means that o_1 could have learned about the existence of o_2! Therefore, such a communication must be deemed illegal. (Cf. also the informal discussion in the introductory Section 1, especially Figure 1).

These considerations have the following two consequences. First and most importantly, the component must *keep track* of which identities it gives away to which object in order to exclude situations as just described. To be able to do so and as a second consequence, the communication with the environment must be labeled in such a way, that sender and receiver are contained in the label. This means we *augment* the programs such that always caller and callee can be transmitted in the communication; in general, this is not the case; for instance in the case of a method call, the caller is anonymous. Therefore, transitions are labeled with the kind of communication, the thread identifier, the transmitted

---

[9] The attentive reader will have noticed that there is another assumption underlying the non-observability of instantiation, namely that there is no bound on the number of objects in the system, i.e., there is no "out-of-heapspace" situation.

values, and in case of calls, the name of the method. The labels are shown in Table 8. To indicate that the caller identity is meant as an augmentation, we write it in brackets: $[o]$. For the return label, caller and callee are not needed.

$$
\begin{aligned}
\gamma ::= {} & n\langle[o]\,call\ o.l(\vec{v})\rangle \mid n\langle return(v)\rangle && \text{basic labels} \\
\mid {} & creates\ o \mid \nu(n{:}T).\gamma \\
a ::= {} & \gamma? \mid \gamma! && \text{send and receive labels}
\end{aligned}
$$

Table 8. Labels

For the transmitted values, the labels further distinguish between a free transmission of a value, i.e., a name already commonly known by the communication partners, and the transmission of fresh names, where the name occurs under the typed ν-binder. For instance, for object creation, the identity o_2 of the freshly instantiated object is always transmitted bound, which is noted in the label as $\nu(o{:}c).creates\ o$. Note that for instantiation, the thread identity and the creator are not needed in the label.

4.3 Connectivity contexts and cliques

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 and thread 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. Besides the relationships amongst objects, we need to keep one piece of information concerning the "connectivity" of *threads*. In order to exclude situations where a known thread leaves the environment 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.

Formally, the semantics of an open component is given by labeled transitions between judgments of the form $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$, where

$$
E_\Delta \subseteq \Delta \times (\Delta + \Theta) \ . \tag{1}
$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$. We will write $n_1 \hookrightarrow n_2$ ("n_1 may know n_2") for pairs from these relations.

In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component. For the formulation of the semantics itself, the commitment contexts E_Θ would not really be needed: It is unnecessary to advertise the approximated E_Θ-commitments to exclude impossible behavior if one has the code of the component at hand to formulate the semantics. Nevertheless, a symmetric situation is advantageous, for instance

if we come to characterize the possible traces of a component independent from its implementation.

As mentioned, the component has to keep track in E_Δ, which objects of the environment are connected, and symmetrically for its own objects in E_Θ. 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.

More technically, the worst case assumptions about the actual situation 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.,

$$\leftrightharpoons \triangleq (\hookrightarrow\downarrow_\Delta \cup \hookleftarrow\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ . \tag{2}$$

Note that we close the part of $\hookrightarrow$ concerning only environment objects from Δ, but not wrt. objects at the *interface,* i.e., the part of $\hookrightarrow \subseteq \Delta \times \Theta$. We will also need the union of $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, for which we will also write $\leftrightharpoons\hookrightarrow$. 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, and sometimes we allow ourselves to write just $E_\Delta \vdash v_1 \leftrightharpoons v_2$, leaving Δ and Θ to be understood from the context.

The relation $\leftrightharpoons$ is an equivalence relation on the objects from Δ and partitions them in 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.

Remark 1 (Thread identifier). As a side remark concerning the use of thread names n in E_Δ: As mentioned, besides connections between objects, E_Δ contains also information about thread names. The stored information about threads is rather restricted, though. In case that the active thread has left the component, the only thing which needs to be remembered is the object *into which* the thread has left the component. Since thread identifiers cannot be stored in variables or communicated in method calls, there are no pairs of the form $p \hookrightarrow n$ in E_Δ, when n is a thread identifier. Also, for each thread name n from Δ, there is at most one pair $n \hookrightarrow o$ in E_Δ, where o is an object reference from Δ. Since, unlike object names, a thread name n can (and will) occur in the domain of Δ and Θ, the disjoint union $\Delta + \Theta$ is not literally true. It holds, however, for object names, which play the crucial role in the development.

Having introduced E_Δ and E_Θ as part of the judgment, we must still clarify what it "means", i.e., when does $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ hold? Besides the typing part, which remains unchanged, this concerns the commitment part E_Θ. The relation E_Θ asserts about the component C that the connectivity of the objects from the component is *not larger than* the connectivity entailed by E_Θ. Given a component C and to object names o_1 from Θ and o_2 from $\Theta + \Delta$, we write $C \vdash o_1 \hookrightarrow o_2$, if $C = C' \parallel o_1[\ldots, f = o_2, \ldots]$, i.e., o_1 contains in one of its fields a reference to o_2. We can thus define:

Definition 1. *The judgment* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ *holds, if* $\Delta \vdash C : \Theta$, *and if* $C \vdash n_1 \hookrightarrow n_2$, *then* $E_\Theta; \Theta \vdash n_1 \leftrightharpoons\hookrightarrow n_2 : \Delta$.

We often simply write $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ to assert that the judgment is satisfied.

Note again, that the pairs listed in a commitment context E_Θ do not require the *existence* of connections in the components, it is rather the contrapositive situation: If E_Θ does *not* imply that two objects are in connection, possibly following the connection of other objects, then they must not be in connection in C. Thus, a larger E_Θ means a weaker specification. To make this precise, let us define what it means for one context to be stronger than another:

Definition 2 (Entailment). $\Delta_1; E_{\Delta_1}; \Theta_1 \vdash \Delta_2; E_{\Delta_2}; \Theta_2$ *iff. for all names* $n \in \Delta_2$ *and* $n' \in \Delta_2 \times (\Delta_2 + \Theta_2)$ *we have: if* $\Delta_1; E_{\Delta_1}; \Theta_1 \vdash n \leftrightharpoons\hookrightarrow n'$, *then* $\Delta_2; E_{\Delta_2}; \Theta_2 \vdash n \leftrightharpoons\hookrightarrow n'$.

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

4.4 External steps

After having clarified the interpretation of the connectivity contexts E_Δ and E_Θ, we can address the external behavior of a component more formally. The external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments and shown in Table 9 for the exchange of free names and in Table 10 dealing with bound names.

To get a general impression, let us first go through the rules ignoring the relational part concerning the E_Δ- and E_Θ-assumptions. For incoming calls, given in rules CALLI_1 and CALLI_2, we need to distinguish whether the calling thread is already resident in the configuration, i.e., whether it is a *reentrant* call wrt. the configuration, or not. In case the thread visits the configuration for the first time (rule CALLI_1), the execution of the method body, terminated by a return, is added in parallel. In case of a reentrant call, a blocked part of the thread is already contained in the configuration (cf. rule CALLI_2), the new method body is "stacked" on top of the prior, blocked part. When the activity of a thread returns to the environment (cf. rule RETURNO), the return-statement is "popped-off" the thread; in combination with the rules for incoming calls we

$$\frac{\begin{array}{c} ; \Delta \vdash o_1 : [\dots] \qquad ; \Delta, \Theta \vdash o_2.l(\vec{v}) : T \qquad o_2 \in \Theta \qquad \Delta \vdash n : thread \qquad n \notin dom(\Theta) \\ \Delta; E_\Delta \vdash [o_1] \leftrightharpoons \hookrightarrow \vec{v} : \Theta \qquad \Delta; E_\Delta \vdash [o_1] \leftrightharpoons \hookrightarrow o_2 : \Theta \qquad \Delta; E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{\Theta} = \Theta, n{:}\,thread \qquad \acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{n\langle [o_1] call\; o_2.l(\vec{v}) \rangle?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle let\; x{:}T = o_2.l(\vec{v})\; in\; return[o_1]\; x \rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}} \;\text{CALLI}_1$$

$$\frac{\begin{array}{c} ; \Delta \vdash o_1 : [\dots] \qquad ; \Delta, \Theta \vdash o_2.l(\vec{v}) : T \qquad o_2 \in \Theta \qquad \Delta \vdash n : thread \\ E_\Delta \vdash [o_1] \leftrightharpoons \hookrightarrow \vec{v} : \Theta \qquad E_\Delta \vdash [o_1] \leftrightharpoons \hookrightarrow o_2 : \Theta \qquad E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{\Theta} = \Theta, n{:}\,thread \qquad \acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle let\; x'{:}T' = [o_2]\; blocks\; for\; o'_2\; in\; t \rangle : \Theta; E_\Theta \xrightarrow{n\langle [o_1] call\; o_2.l(\vec{v}) \rangle?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle let\; x{:}T = o_2.l(\vec{v})\; in\; return[o_1]\; x; let\; x'{:}T' = [o_2]\; blocks\; for\; o'_2\; in\; t \rangle : \Theta; \acute{E}_\Theta \end{array}} \;\text{CALLI}_2$$

$$\frac{\acute{E}_\Delta = E_\Delta + ([o_1] \hookrightarrow v, n \hookrightarrow [o_1]) \qquad \acute{E}_\Theta = E_\Theta \setminus n}{\Delta; E_\Delta \vdash C \parallel n\langle let\; x{:}T = return[o_1]\; v\; in\; t \rangle : \Theta; E_\Theta \xrightarrow{n\langle return(v) \rangle!} \Delta; \acute{E}_\Delta \vdash C \parallel n\langle t \rangle : \Theta; \acute{E}_\Theta} \;\text{RETO}$$

$$\frac{o_2 \in \Delta \qquad \acute{E}_\Delta = E_\Delta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \qquad \acute{E}_\Theta = E_\Theta \setminus n}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle let\; x{:}T = [o_1]\; o_2.l(\vec{v})\; in\; t \rangle : \Theta; E_\Theta \xrightarrow{n\langle [o_1] call\; o_2.l(\vec{v}) \rangle!} \\ \Delta; \acute{E}_\Delta; \vdash C \parallel n\langle let\; x{:}T = [o_1]\; blocks\; for\; o_2\; in\; t \rangle : \Theta; E_\Theta \end{array}} \;\text{CALLO}$$

$$\frac{\begin{array}{c} ; \Delta, \Theta \vdash v : T \qquad \Delta; E_\Delta \vdash o_2 \leftrightharpoons \hookrightarrow v : \Theta \qquad \Delta; E_\Delta \vdash n \hookrightarrow o_2 : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{E}_\Theta = E_\Theta + (o_1 \hookrightarrow v, n \hookrightarrow [o_1]) \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle let\; x{:}T = [o_1]\; blocks\; for\; o_2\; in\; t \rangle : \Theta; E_\Theta \xrightarrow{n\langle return(v) \rangle?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle t[v/x] \rangle : \Theta; \acute{E}_\Theta \end{array}} \;\text{RETI}$$

$$\frac{c \in \Delta}{\begin{array}{c} \Delta; E_\Delta \vdash n\langle let\; x{:}T = new\; c\; in\; t \rangle : \Theta; E_\Theta \xrightarrow{\nu(o_3:c).creates o_3 !} \\ \Delta, o_3{:}c; E_\Delta \vdash n\langle let\; x{:}T = o_3\; in\; t \rangle : \Theta; E_\Theta \end{array}} \;\text{NEWO}$$

$$\frac{C(c) = [\![O]\!] \qquad c \in \Theta \qquad E_\Delta \vdash \acute{E}_\Delta \downarrow_{\Delta \times (\Delta + \Theta)}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu(o_3:c).creates o_3 ?} \Delta; \acute{E}_\Delta \vdash C \parallel o_3[O] : o_3{:}c, \Theta; E_\Theta} \;\text{NEWI}$$

Table 9. External steps

see that the remaining part of the thread remains blocked. Similarly, *calling* an external object leaves the local execution in a blocked state, waiting for the matching return carrying the returned value (cf. rule CALLO and RETURNI). Note that the name context Δ is used to distinguish an external call in rule CALLO from an internal one which is covered by the corresponding rule from Table 5.

As for E_Δ resp. E_Θ and the relationship of communicated values, the incoming and outgoing communication play dual roles. Remember that the relation E_Δ contains pairs of objects (and thread names) from Δ as well as from Θ, and dually for E_Θ. In general, E_Θ overapproximates the actual connectivity of the component, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values.

Now for the *incoming* call both in rule $\textsc{CallI}_1$ and $\textsc{CallI}_2$, we require that the caller o_1 be acquainted with the callee o_2 and with all of the (object reference) arguments. Furthermore it must be checked that the incoming thread originates from a group of objects in connection with the one to which the thread had left the component the last time: $\Delta; E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta.$[10]

To be able to assure these connectivity conditions, the identity of the callee has been remembered as part of the block-syntax when the call was issued. It is worth mentioning that in rule $\textsc{ReturnI}$ 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 can that the thread is acquainted with the callee. If fact, both requirements will be automatically assured for traces where calls and return occur in correct manner.

A commonality for incoming communication 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 $E_\Delta \setminus n$

Outgoing communication, on the other hand, does not impose restrictions as premise; instead it *extends* the pool of assumption E_Δ by adding communicated names. After an outgoing call, for instance, it is assumed that the callee knows all the arguments it has received and furthermore that the thread now knows the callee-identity (cf. rule $\textsc{CallO}$). A analogous extension of E_Δ is done for outgoing returns in rule $\textsc{ReturnO}$, except that the caller as the receiver of the return communication is anonymous as far as the label is concerned.

The last two rules $\textsc{NewO}$ and $\textsc{NewI}$ handle instantiation across the component boundary. In the first case, the instantiation inside the configuration refers to a class available in the environment, i.e., for a name $c \in \Delta$. The reference to the new object is kept locally in the creator thread and the identity and its type is communicated to the outside, where the label $\nu(o_3{:}c).creates\ o_3!$ indicates that the identity is assumed to be fresh.[11] Dually, $\textsc{NewI}$ allocates a new object when requested from outside by looking up the code of the object; in the rule $C(c) = [\![O]\!]$ abbreviates $C \equiv c[\![O]\!] \parallel C'$; that C is of this form is assured by the static type system. Unlike the situation in the object calculus, object creation can have an external effect, namely when a class outside the current configuration is instantiated.

Another phenomenon not encountered so far is the fact that for an incoming instantiation in rule $\textsc{NewI}$, a *new* identity is transmitted, i.e., an identity received by bound transmission. The proviso of $\textsc{NewI}$ dealing with the extension of E_Δ to $\acute{E}_\Delta$ is discussed together with the rules for bounded communication to which we come after a short digression.

Before looking at the rules dealing with the ν-binders, the following technical side remark discusses in more detail the formulation of the rules for incoming calls and the role of the caller therein.

---

[10] Since the caller o_1 is in the domain of Δ, we can write $n \leftrightharpoons [o_1]$ instead of $n \leftrightharpoons\hookrightarrow [o_1]$.
[11] Later, this step will be *unobservable.*

Remark 2 (Caller identity). Note that in the situation of rules CALLI_i, the caller o_1 might be unknown to the component so far, which means that it is introduced in the call-step to the component via scope extrusion with the help of the rules of Table 10 below. Since in the step for the incoming call, the identity of o_1 is arbitrary except that o_1 must be connected to the callee, the arguments, and the thread, of course, and since furthermore ultimately the identity of the caller is ignored in the trace semantics, one could imagine also the following variant of the rule: leaving out o_1 in the call-labels, the proviso of the CALLI-rules can be equivalently written as

$$\exists o_1.\ \Delta; E_\Delta \vdash o_1 \leftrightharpoons\hookrightarrow \vec{v} : \Theta \wedge \Delta; E_\Delta \vdash o_1 \leftrightharpoons\hookrightarrow o_2 : \Theta \wedge \Delta; E_\Delta \vdash n \leftrightharpoons\hookrightarrow o_1 : \Theta \ ,$$

which perhaps expresses the intuition more clearly: for an incoming call to be possible, there must *exist* some caller, whose identity does not play a role and which is appropriately connected. In effect, the existential quantification would then range in some sense over the objects already known from Δ and those (yet) *unknown.*

Why are then the callers o_1 mentioned in the label and thus participating in the dynamic scoping mechanism of the semantics instead of taking the solution of the above existential formula? First the ν-binder has a flavor of existential quantification, therefore the scoping mechanism can handle the issue.

Besides that, the technical reason for the decision to augment the traces and the programs with the caller identity has to do with the necessary bookkeeping of the connectivity contexts, especially for *returns.* When the corresponding outgoing return for the call under consideration happens, it is clear that the call returns to the *same* object o_1 who issued the call. Whether the caller is included in the label or treated by existential quantification, in any case an element of guessing is involved in rules CALLI_i, but it is crucial, that in order to update E_Δ in rule RETURNO appropriately, one needs to *remember* the choice o_1 taken at *call-time.* The cleanest way to remember it is to record the pick in the trace and rely on the scoping mechanism of the ν-binders.

Note that the type or interface of the caller plays (almost) no role: as long as the object has a method at all, it can *issue* a call, since the interface speaks only about the ability of *accepting* calls.   □

Building upon these rules, the ones from Table 10 take additionally scoping information into account. The rules deal with calls and returns, only, as the special case of instantiation has been handled already in Table 9.

The ν-binder for configurations influence only names occurring freely in the label (cf. rule COMM). In case of a *bound input* (cf. rule BIN), the name's scope is extruded *into* the component and therefore the step of the premise is checked in the extended assumption $\Delta, n{:}T$. The treatment of the knowledge base E_Δ requires a closer look. When the new name n is introduced to the component, it is from then on part of the known names, and the component must estimate the possible acquaintances of n. In general, the new name of an object will be

$$\frac{n \notin fn(a) \qquad \Delta; E_\Delta \vdash C : \Theta, n{:}T; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, n{:}T; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \nu(n{:}T).\acute{C} : \acute{\Theta}; \acute{E}_\Theta \setminus n} \text{ COMM}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta; E_\Delta \vdash \Delta, n{:}T; E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta' \\ \Delta, n{:}T; E'_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu(n{:}T).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \text{ BIN}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta; E_\Delta \nvdash T : [\![\dots]\!] \\ \Delta; E_\Delta \vdash C : \Theta, n{:}T; E_\Theta + E(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{\nu(n{:}T).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \text{ BOUT}$$

Table 10. External steps (scoping for calls and returns)

related to *some* other objects, the question only is, to which ones? As the name is fresh to the component and nothing is known yet about the connectivity of the object, the component may just *guess* to which ones the new object belongs. This means, the rule non-deterministically extends the assumptions E_Δ to E'_Δ by adding pairs $n' \hookrightarrow n$, where n is the new identity.

Now the gist is to understand that while the component may guess which acquaintances the new object has, it is not completely free to do so! Since E_Δ is maintained as a worst-case assumption about the connectivity of the known external objects, learning about the existence of a fresh object must not invalidate this assumption. Intuitively, by creating new objects, which are initially unknown to the component, the environment cannot contact objects it could not contact otherwise. This restriction is captured in the proviso

$$\Delta; E_\Delta \vdash \Delta'; E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta \ ,$$

where $\Delta' = \Delta, n{:}T$ in the rule, which requires that the addition of connectivity of the new identity n added to Δ may not lead to new derivable equations for the objects previously known. The requirement, $\Delta; E_\Delta \vdash E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta$ thus stands for the implication: If $\Delta'; E'_\Delta \vdash o_1 \leftrightharpoons \hookrightarrow o_2 : \Theta$, then $\Delta; E_\Delta \vdash o_1 \leftrightharpoons \hookrightarrow o_2 : \Theta$, for all o_1 from Δ and o_2 from Δ, Θ. In other words, E'_Δ is a *conservative* extension of E_Δ wrt. the old objects. For illustration, see also Example 2 below.

Remains the case for bound output, where the scope of the name is extruded, marking it as bound on the label. The corresponding rule BOUT, and especially the transition in the premise is checked with the extended commitment $\Theta, n{:}T$. For the connectivity, we add *all* names from $\Theta' \times (\Theta' + \Delta)$ where $\Theta' = \Theta, n{:}T$, which according to C are acquainted with n. We write $E(C, n)$ for that set of names.

Note that the operational rules do not contain rules for communication or synchronization. The reason is that we are not interested in axiomatizing the behavior of parallel composition in general, but are content to characterize the

external behavior of *one* parallel component. It would be rather straightforward to add such synchronization rules, though, folding matching pairs of communication steps into a τ-step, respectively a $\rightsquigarrow$-step in case of instantiation.

4.5 Examples

The following two short examples illustrate the transition semantics for receiving a bound value, especially the interplay of the basic rules of Table 9 and those treating scoping from Table 10. The second example especially shows the role the conservative guessing of connectivity of new incoming names.

Example 1. We take the label $\nu(o_3:c_3).\nu([o_1]:c_1).t\langle[o_1]\,call\ o_2.l(o_3)\rangle?$, where both caller and the argument are bound. In the derivation, Δ' and Δ'' represent extended assumption contexts, i.e., $\Delta' = \Delta, o_3:c_3$ and $\Delta'' = \Delta', o_1:c_1$. Similarly, the extended relation E_Δ'' equals $E_\Delta'' = E_\Delta, o_1 \hookrightarrow o_2, o_1 \hookrightarrow o_3$ and moreover, $E_\Delta''' = E_\Delta'' \setminus n$. In the rules we concentrate on the connectivity information and elide premises dealing with well-typedness of the communicated arguments. Since furthermore we leave internal structure of C unspecified, we also leave the form of the commitment context E_Θ and its variants unspecified.

$$\cfrac{\cfrac{\Delta''; E_\Delta'' \vdash [o_1] \leftrightharpoons o_3 : \Theta \quad \cfrac{o_2 \in \Theta \quad o_1 \in \Delta''}{\Delta''; E_\Delta'' \vdash [o_1] \leftrightharpoons\hookrightarrow o_2 : \Theta} \quad \Delta''; E_\Delta'' \vdash n \leftrightharpoons [o_1] : \Theta}{\Delta''; E_\Delta'' \vdash C : \Theta; E_\Theta \xrightarrow{n\langle[o_1]call\ o_2.l(o_3)\rangle?} \Delta''; E_\Delta''' \vdash C' : \Theta'; E_\Theta'} \text{CALLI}_i}{\cfrac{\Delta'; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu([o_1]:c_1).n\langle[o_1]call\ o_2.l(o_3)\rangle?} \Delta''; E_\Delta''' \vdash C' : \Theta'; E_\Theta'}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu(o_3:c_3).\nu([o_1]:c_1).n\langle[o_1]call\ o_2.l(o_3)\rangle?} \Delta''; E_\Delta''' \vdash C' : \Theta'; E_\Theta'} \text{BIN}} \text{BIN}$$

Swapping the order of the binding occurrences for caller and argument in the label yields a slightly different derivation; here $\Delta' = \Delta, [o_1]:c_1$ and $\Delta'' = \Delta', o_3:c_3$, and for the relation $E_\Delta' = E_\Delta, o_1 \hookrightarrow o_2$ and $E_\Delta'' = E_\Delta', o_1 \hookrightarrow o_3$:

$$\cfrac{\cfrac{\Delta''; E_\Delta'' \vdash [o_1] \leftrightharpoons o_3 : \Theta \quad \cfrac{o_2 \in \Theta \quad o_1 \in \Delta''}{E_\Delta'' \vdash [o_1] \leftrightharpoons\hookrightarrow o_2 : \Theta} \quad \Delta''; E_\Delta'' \vdash n \leftrightharpoons [o_1] : \Theta}{\Delta''; E_\Delta'' \vdash C : \Theta; E_\Theta \xrightarrow{n\langle[o_1]call\ o_2.l(o_3)\rangle?} \Delta''; E_\Delta''' \vdash C' : \Theta'; E_\Theta'} \text{CALLI}_i}{\cfrac{\Delta'; E_\Delta'; \vdash C : \Theta; E_\Theta \xrightarrow{\nu(o_3:c_3).n\langle[o_1]call\ o_2.l(o_3)\rangle?} \Delta''; E_\Delta'''; \vdash C' : \Theta'; E_\Theta'}{E_\Delta; \Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu([o_1]:c_1).\nu(o_3:c_3).n\langle[o_1]call\ o_2.l(o_3)\rangle?} \Delta''; E_\Delta''' \vdash C' : \Theta'; E_\Theta'} \text{BIN}} \text{BIN}$$

<div align="right">□</div>

Example 2. Assume two cliques of objects in the environment, objects o_1^i and objects o_3^j, i.e., $\Delta; E_\Delta \vdash o_1^i \leftrightharpoons o_1^j : \Theta$ and $\Delta; E_\Delta \vdash o_3^i \leftrightharpoons o_3^j : \Theta$, but $\Delta; E_\Delta \not\vdash o_1^i \leftrightharpoons o_3^j : \Theta$. Let now the component do an incoming communication step learning about a new identity, o_0 say, as the calling object.

$$\cfrac{\cfrac{o_2 \in \Theta \quad ; \Delta' \vdash o_0 : [\ldots] \quad \Delta'; E_\Delta' \vdash [o_0] \leftrightharpoons\hookrightarrow o_2 : \Theta}{E_\Delta'; \Delta' \vdash C : \Theta; E_\Theta \xrightarrow{n\langle[o_0]call\ o_2.l_1()\rangle?} E_\Delta'; \Delta' \vdash C' : \Theta'; E_\Theta'} \quad \Delta; E_\Delta \vdash \Delta'; E_\Delta' \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu([o_0]:c_1).n\langle[o_0]call\ o_2.l_1()\rangle?} \Delta'; E_\Delta' \vdash C' : \Theta'; E_\Theta'}$$

Assume that E_Δ is extended into $E'_\Delta = E_\Delta, o_0 \hookrightarrow o_2, o_0 \hookrightarrow o_1^1$, i.e., guessing that o_0 knows (what turns out in the subderivation step to be) the callee o_2 as well as o_1^1, for instance (one could have taken any subset of the o_1^i or alternatively of the o_3^j). This guess allows to derive as second transition step, that the meanwhile known object o_0 —at least known by way of augmentation— calls o_2 again and this time passes o_1^2, say, as parameter:

$$\frac{o_2 \in \Delta' \qquad \Delta', E'_\Delta \vdash [o_0] \leftrightharpoons\hookrightarrow o_2 : \Theta' \qquad \Delta'; E'_\Delta \vdash [o_0] \leftrightharpoons\hookrightarrow o_1^2 : \Theta'}{\Delta'; E'_\Delta \vdash C' : \Theta'; E'_\Theta \xrightarrow{n'\langle[o_0]call\ o_2 . l_1(o_1^2)\rangle?} E''_\Delta; \Delta' \vdash C'' : \Theta''; E''_\Theta}$$

As mentioned above, instead of guessing $o_0 \hookrightarrow o_1^i$ in the first step, one might as well have chosen $o_0 \hookrightarrow o_3^j$ instead (except that in that case the second step would not have been derivable).

What would *not* be accepted as legal BIN-step is, that the relation E'_Δ contains *both* $o_0 \hookrightarrow o_1^i$ and $o_0 \hookrightarrow o_3^j$ pairs. This would mean that by receiving the new identity o_0, suddenly the objects o_1^i and o_3^j are connected in the eyes of the component. This is impossible, however, since the component makes a worst-case assumption about the known objects, and object creation in the environment cannot be used to merge cliques of objects. Formally, the wrong guess would mean that $\Delta'; E'_\Delta$ is not a conservative extension of $\Delta; E_\Delta$, since $\Delta'; E'_\Delta \vdash o_1^i \leftrightharpoons o_3^j : \Theta$ but $\Delta; E_\Delta \nvdash o_1^i \leftrightharpoons o_3^j : \Theta$. $\qquad\qquad\square$

4.6 Trace semantics

This section contains the denotational semantics for well-typed components, which is, as in the object-based setting, a trace semantics, containing all sequences of external steps of the program fragment. On the surface, the trace semantics presented here looks quite similar to the one for the object based setting.

One obvious change compared to the object-based framework are the labeled steps for cross-border instantiation *creates o_2?* and *creates o_2!* (plus ν-bindings). Another, more superficial difference is that, to keep track of the connectivity, we were led to incorporate also the caller into the labels for method calls. A final difference does not refer to the form of a single trace, but to the *set* of all traces of a component, for which we will write $[\![\Delta; E_\Delta \vdash C : \Theta; E_\Theta]\!]_{trace}$. In all but degenerated cases, a component does not possess one single trace, but an (infinite) set of traces.[12] Crucial for the semantics (or any trace semantics) are its *closure* properties, which characterize, given a set of traces in $[\![\Delta; E_\Delta \vdash C : \Theta; E_\Theta]\!]_{trace}$, which traces are necessarily included, too. Abstractly speaking, the new situation that the environment can instantiate isolated cliques of objects in the component, or from a dual perspective, that the observing context may

---

[12] The *only* case is an component where no object has a method, no activity of its own, and with especially no classes containing methods, since otherwise the trace set contains incoming calls.

may not be able to coordinate all its observations makes the closures larger. Again from the dual perspective of the observer, it increases the "uncertainty of observation" and leads thus to a coarser notion of observational preorder (although depending on the connectivity contexts). But first we get rid of the create-labels in the semantics.

4.6.1   Lazy instantiation In the operational semantics of Section 4, instantiations across the component boundary are visible. Without constructor methods, however, it is clear, that the instantiation alone and the fact that an object is existent in the environment cannot be used by an observer. The only way to do observations is by method calls. Therefore, a fully abstract trace semantics must not contain the labels $createso_2!$ and $createso_2?$

In this section we present, as intermediate step, a variant of the external step semantics from Table 9 and Table 10 where object creation across the component boundary is not visible. Instead, new objects are incorporated only at the point when they are first communicated to the other side or used from the other side. To distinguish the two forms of semantics, we call the one from Section 4 the semantics with *eager instantiation* or in short early semantics, the one presented here the semantics with *lazy instantiation* or late semantics.

As in the early operational semantics, there are rules dealing with the core labels without binding, and rules for scoping. For the core labels, the rules for the late semantics are *identical* to those from Table 9 except that the rules for incoming and outgoing object instantiation NEWI and NEWO are missing. Instead, the set of rules is extended by the following rule NEWO_{lazy}:[13]

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

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.

The fact that in the above rule the object is *not* yet created means that the interpretation of the ν-binder slightly changes. Up to now, and as mentioned in connection with the type system in Section 3, a term $\nu n{:}T.C$ could be interpreted as the assertion that C contains the named entity n, where n refers to an object, a class, or a thread. With lazy instantiation and in presence of NEWO_{lazy}, this no longer holds, since the execution of the *new*-statement reserves a new name for the object, but does not immediately creates the object itself. Related to that,

---

[13] Doing a $\rightsquigarrow$-step, the rule would seems to fit well into the internal steps. Nevertheless, we consider it as a step between typing judgments, as the step relies on the environmental information that the appropriate class c is externally available.

the typing rule T-Nu for the ν-binder requires a small refinement. In case the binder refers to an object whose class is a class of the environment and which which consequently will reside itself in the environment, the binding is to be added to the assumption context Δ and not to Θ.

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

Remark 3 (Lazy instantiation). The fact that in the presence of lazy instantiation the ν-binder does not assert the existence of the named entity might seem strange. Note, however, that some alternatives to the rule NewO_{lazy} and the refinement of the type system do not work. For instance, it is not possible to change rule NewO_{lazy} in such a way that not only the new name is introduced but also the object is already created to be kept in pristine condition until it is needed and then to *copy* it to the environment. By simultaneously creating the name and instantiating the object, the ν-binder would stipulate the existence of a corresponding object. However, the class being instantiated is represented in the assumption context only in form of its type or interface; thus there is not enough information to create an instance of the external class inside the component .

Another tempting alternative would be to not use the ν-construct at all, but directly put the binding $o_3{:}c$ from rule NewO_{lazy} into the assumption context. That's not possible, either, as in this case the necessity for the environment to actually create an object when it is needed, is unnoticed. □

Besides these changes, the scoping rules in the late semantics contain two additional rules to deal with new objects, BOut_{new} and BIn_{new} (cf. Table 11). They take care that objects whose reference has been created locally in the component are finally created in the environment, where they belong, the first time there name is exported; dually for instances of component classes instantiated by the environment.

The first case corresponds to rule BOut_{new}, where unlike the situation in BOut, the *assumption* context is extended by the new name. Note here that the new object is *unknown* in the environment at the current stage; therefore E_Δ is not extended in the judgement prior to the transition in the premise. In contrast, the name o might already be passed around internally, which means that E_Θ must be extended in the same way as in rule BOut.

For BIn_{new}, the situation is dual. Remember that the connectivity context E_Δ contains also pairs from $\Delta \times \Theta$. Therefore, it has to be checked also here that E'_Δ is a conservative extension of E_Δ.

The following example illustrates the working of lazy instantiation.

Example 3. Assume that c_2 and c_3 are two external classes, i.e., $c_2, c_3 \in \Delta$. Let further o_1 be a component object, which is already known to the environment,

$$\frac{n \notin fn(a) \qquad \Delta; E_\Delta \vdash C : \Theta, n{:}T; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, n{:}T; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \nu(n{:}T).\acute{C} : \acute{\Theta}; \acute{E}_\Theta \setminus n} \text{ Comm}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta; E_\Delta \vdash \Delta, n{:}T; E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta' \\ \Delta, n{:}T; E'_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu(n{:}T).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \text{ BIn}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta; E_\Delta \nvdash T : [\![\dots]\!] \\ \Delta; E_\Delta \vdash C : \Theta, n{:}T; E_\Theta + E(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{\nu(n{:}T).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \text{ BOut}$$

$$\frac{\begin{array}{c} o \in fn(\gamma) \qquad \Theta \vdash c : [\![\dots]\!] \qquad C(c) = [\![O]\!] \\ \Delta; E'_\Delta \vdash C : \Theta, o{:}c; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \quad \Delta; E_\Delta \vdash \Delta; E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\nu(o{:}c).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} \parallel o[O] : \acute{\Theta}; \acute{E}_\Theta} \text{ BIn}_{new}$$

$$\frac{\begin{array}{c} o \in fn(\gamma) \qquad \Delta \vdash c : [\![\dots]\!] \\ \Delta, o{:}c; E_\Delta \vdash C : \Theta; E_\Theta + E(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(o : c).C : \Theta; E_\Theta \xrightarrow{\nu(o{:}c).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \text{ BOut}_{new}$$

Table 11. External steps (scoping, late)

i.e., $\Theta \vdash o_1 : c_1$ for some class c_1 in Θ. The following steps describe a situation, where a thread creates an object of the external class c_2, and calls a method of this object. Furthermore, it passes to the callee o_2 the reference to another object o_3, which likewise is an instance of an external class, namely c_3 and which is unknown to the environment yet:

$$
\begin{array}{ll}
1 & \Delta; E_\Delta \vdash \nu(o_3{:}c_3).n\langle let\ x{:}c' = new\ c'\ in[o_1]\ x.l(o_3); t\rangle\ :\ \Theta; E_\Theta \rightsquigarrow \\
2 & \Delta; E_\Delta \vdash \nu(o_3{:}c_3, o_2{:}c_2).n\langle let\ x{:}c' = o_2\ in[o_1]\ x.l(o_3); t\rangle\ :\ \Theta; E_\Theta \rightsquigarrow \\
3 & \Delta; E_\Delta \vdash \nu(o_3{:}c_3, o_2{:}c_2).n\langle[o_1]\ o_2.l(o_3); t\rangle\ :\ \Theta; E_\Theta \xrightarrow{\nu(o_2{:}c_2, o_3{:}c_3).n\langle[o_1] call\ o_2.l(o_3)\rangle!} \\
4 & \Delta, o_2{:}c_2, o_3{:}c_3; E'_\Delta \vdash n\langle[o_1]\ blocks\ for\ o_2; t\rangle\ :\ \Theta; E_\Theta
\end{array}
$$

After some internal steps, both o_2 and o_3 escape to the environment by lazy instantiation (rule NEWO$_{lazy}$).

As mentioned, in the example we have chosen o_1 to be an externally known internal instance, while the argument o_3 and the callee o_2 are not yet published and exported external instances. Note that the rules does not allow an analogous step in the situation, where also the caller o_1 is an instance of an external class.

4.6.2 Traces A trace of a well-typed component is a sequence of external steps in the late semantics, i.e., not containing instantiation labels. The corresponding rules are given in Table 12. We write $[\![\Delta; E_\Delta \vdash C : \Theta; E_\Theta]\!]_{trace}$ for the set of traces of a component $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$.

$$
\frac{C \Longrightarrow C'}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{\epsilon}{\Longrightarrow} \Delta; E_\Delta \vdash C' : \Theta; E_\Theta}\ \text{INTERNAL}
$$

$$
\frac{\Delta_1; E_{\Delta_1} \vdash C_2 : \Theta_1; E_{\Theta_1} \stackrel{a}{\longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}}{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{a}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}}\ \text{BASE}
$$

$$
\frac{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{s_1}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2} \stackrel{s_2}{\Longrightarrow} \Delta_3; E_{\Delta_3} \vdash C_3 : \Theta_3; E_{\Theta_3}}{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{s_1 s_2}{\Longrightarrow} \Delta_3; E_{\Delta_3} \vdash C_3 : \Theta_3; E_{\Theta_3}}\ \text{CONC}
$$

Table 12. Traces

Except that cross-border instantiation is postponed, the late and the early semantics coincide. We write $\stackrel{s}{\Longrightarrow}_l$ when referring to traces according to the late semantics, and $\stackrel{s}{\Longrightarrow}_e$ to those of the early semantics. Analogously $[\![\_]\!]^l_{trace}$ denotes the set of late traces of a component, and $[\![\_]\!]^e_{trace}$ the set of early traces, where the instantiation steps are *removed*.

Lemma 1 (Late = early). *Assume $\Delta, E_\Delta \vdash C : \Theta$. Then $[\![\Delta; E_\Delta \vdash C : \Theta]\!]^l_{trace} = [\![\Delta; E_\Delta \vdash C : \Theta]\!]^e_{trace}$*

Proof. Lengthy but straightforward. □

5   Conclusion

In this report we presented, as an extension of the work of [3], an operational
semantics and a trace semantics of a class-based, object-oriented calculus with
multithreading. The seemingly innocent step from an *object-based* setting as in
[3] 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 may therefore be instructive, to review the differences in this conclusion,
to explain them from a higher perspective, especially trying to understand how
the result of [3] can be understood as a special case of the framework explored
here.

The fundamental dichotomy underlying the observational definition of equiv-
alence is the one between the inside and the outside: program or component vs.
environment or observer. This leads to the crucial difference between object-
based languages, instantiating from objects, and class-based language, instanti-
ating from classes: In the class-based setting, instantiation may *reach across 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 compli-
cations, expounded here at some great length, follow from this difference. The
most visible complication is 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 [3], 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), never new isolated cliques are
created. The object-based case can therefore be understood where invariantly
(and trivially) $E_\Delta = \Delta \times (\Delta + \Theta)$, while in our setting, we take into account
that E_Δ may be more specific.

We see this study of the semantics as a step towards a full-abstraction result
for the class-based calculus, on which we will report separately, once the details
are hammered out.

Other future work is to extend the language and the semantics in a number
of ways. One inherent feature of the calculus is that objects are *input enabled.*
This disallows to model directly *synchronized methods* as in *Java.* The extension
of the language and should be comparatively mild; the detailed adaptation of
the semantics and the characterization of the legal traces may still be tricky.
Another interesting but non-trivial generalization is to consider *cloning* of ob-
jects, i.e., to create a replica of an object, identical to the original one up-to
the object's identity. In a certain way, *instantiation* of a class is just like cloning
with the restricting that only objects in their initial state can be obtained by the
operation, while cloning can be applied to an object in mid-life. The ability to
create an object in a state different from the initial one makes new observations

possible, most notably the branching structured gets exposed. One therefore has to generalize the *linear-time* framework of traces to a *branching-time* view. We are currently working on the generalization of our results in this respect.

Even more challenging is to take serious the notion of classes in that they are not only considered as generator of new objects by instantiation, but also as template for new classes, i.e., to consider inheritance and subtyping. This makes new "observations" on classes possible, namely by subclassing.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. 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.
3. 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.
4. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
5. 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.
6. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

A   Operational semantics: ν-binders with connectivity

This section sketches an alternative presentation of the late operational semantics from Section 4. One particular part of the definition is disappointing, namely the fact that connectivity information is contained in the contexts E_Δ and E_Θ, but the ν-binders inside the component contain only the name of the entity, but not its connectivity. In order to achieve subject reduction, as a consequence, we where forced to *recheck* the connectivity of the component each time a new name is exported to the environment (cf. Rule BOUT in Table 10).

In order to remedy this, we extend the calculus such that the ν-binder contains connectivity information, as well. The main requirement in this context is, that this information must be *stable* under internal reduction. Without this

stability, a local, context-free definition of the internal reduction relation similar to the one from Table 5 would not be possible, since local reduction steps could then influence encapsulating ν-constructs.

One part of the connectivity information which remains invariant is who created whom. This information suffices to get an accurate picture of the relationship between the objects, since the graph structure which evolves by exchanging values between objects can only connect parts of the object structure which are related by descendence: no object starts its life on its own and them contacts the rest of the objects "out of the blue".

To represent the connectivity, the syntax is extended such that

$$\nu(\Theta, E_\Theta).C$$

describes a component with local names Θ and connectivity E_Θ of the communicated names.

The syntax is further augmented to record the creator of a new instance: the instantiation statement takes now the form

$$let\ x{:}c = [self]\ new\ c\ in\ t$$

where *self* is the ς-bound self-parameter of the corresponding method. Furthermore, the grammar for components (cf. Table 2) is extended to include the production $C ::= n \hookrightarrow n$.

In the internal steps from Table 5, the rule for internal object creation $\textsc{NewO}_i$ is replaced by the following one, which additionally to the instantiation of the object also adds the fact that the creator now knows (or may know) the new instance to the component:

$$c[\![O]\!] \parallel n\langle let\ x{:}c = [o_1]\ new\ c\ in\ t\rangle \rightsquigarrow c[\![O]\!] \parallel \nu(o{:}c).(o_1 \hookrightarrow o \parallel o[O] \parallel n\langle let\ x{:}c = o\ in\ t\rangle)$$

The rules for structural congruence $\equiv$ are unchanged as are the ones for reduction modulo congruence (Table 6 and Table 7). Exporting and importing now also connectivity information, the form of the labels is extended. Instead of having the operational rules guessing the connectivity, the labels are carry now explicit information. This, the binding of a label now takes the form:

$$\nu(\Delta, E_\Delta).\gamma\ , \tag{3}$$

where in case of an incoming communication, we use as usual $\nu(\Theta, E_\Theta).\gamma$ and its syntactic variants.

The rule $\textsc{NewO}_{lazy}$ for lazy instantiation of an external class is adapted analogous to rule $\textsc{New}_i$:

$$c \in \Delta$$

$$\Delta; E_\Delta \vdash n\langle let\ x{:}c = [o_1]\ new\ c\ in\ t\rangle : \Theta; E_\Theta \rightsquigarrow$$

$$\Delta; E_\Delta \vdash \nu(o_3{:}c).o_1 \hookrightarrow o_3 \parallel n\langle let\ x{:}c = o_3\ in\ t\rangle : \Theta; E_\Theta$$

The most visible changes concern the external steps exchanging scope information, of course.

$$\frac{n \notin \mathit{fn}(a) \qquad \Delta; E_\Delta \vdash C : \Theta, n{:}T \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, n{:}T; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta \xrightarrow{a} \Delta'; E'_\Delta \vdash \nu(n{:}T).C' : \acute{\Theta}; \acute{E}_\Theta \setminus n} \ \textsc{Comm}$$

$$\frac{\begin{array}{c} n \in \mathit{fn}(\gamma) \qquad \Delta' = \Delta, n{:}T \quad E'_\Delta = E_\Delta + \tilde{E}_\Delta \quad \Delta; E_\Delta \vdash \Delta'; E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta \\ \Delta'; E'_\Delta \vdash C : \Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta \xrightarrow{\nu(n{:}T; \tilde{E}_\Delta).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \ \textsc{BIn}$$

$$\frac{n \in \mathit{fn}(\gamma) \qquad \Delta; E_\Delta \nvdash T : [\![\ldots]\!] \\ \Delta; E_\Delta \vdash (C \setminus n) : \Theta, n{:}T; E_\Theta + E_\Theta(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{\nu(n{:}T; E_\Theta(C,n)).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \ \textsc{BOut}$$

$$\frac{\begin{array}{c} o \in \mathit{fn}(\gamma) \qquad \Theta \vdash c : [\![\ldots]\!] \qquad C(c) = [\![O]\!] \\ \Theta' = \Theta, o{:}c \quad E'_\Delta = E_\Delta + \tilde{E}_\Delta \qquad \Delta; E_\Delta \vdash \Delta; E'_\Delta \downarrow_{\Delta \times (\Delta + \Theta)} : \Theta \\ \Delta; E'_\Delta \vdash C : \Theta'; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta \xrightarrow{\nu(o{:}c; \tilde{E}_\Delta).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} \parallel o[O] : \acute{\Theta}; \acute{E}_\Theta} \ \textsc{BIn}_{new}$$

$$\frac{o \in \mathit{fn}(\gamma) \qquad \Delta \vdash c : [\![\ldots]\!] \\ \Delta, o{:}c; E_\Delta \vdash C : \Theta; E_\Theta + E_\Theta(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(o : c).C : \Theta; E_\Theta \xrightarrow{\nu(o{:}c; E_\Theta(C,n)).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta} \ \textsc{BOut}_{new}$$

Table 13. External steps, scoping

The rule COMM when the binding does not affect any free name of the label remains unchanged. When a scope opens, making a name known to the environment (cf. rule BOUT), we export also its connectivity to the environment. Unlike the treatment of the corresponding rule in Table 11, the rule here can use the connectivity information kept directly as $\hookrightarrow$-pairs in the component. When exporting the name n, it is crucial that not only all pairs $n \hookrightarrow n'$ and $n' \hookrightarrow n$ with $n' \in \Theta$, are exported, but all names in Θ related to n via the closure $\leftrightharpoons$ resp. $\leftrightharpoons\hookrightarrow$: if E_Θ end up with less connectivity as the objects (and threads) of the component actually realize, *subject reduction* breaks. In abuse of notation we write $E(C, n)$ for this set. Note that unlike in rule the rule from Table 11, $E(C, n)$ or $E_\Theta(C, n)$ here only uses the "parallel entities" $n_1 \hookrightarrow n_2$ of the component and does not refer to the named objects themself.

Having exported the connectivity information for n to the context by extending E_Θ to $E_\Theta + E_\Theta(C, n)$, there is no need in keeping the information inside C as

well. Therefore we remove all information concerning n from the component, for which we write $C \setminus n$. Note that the object or the thread itself is not removed, of course, only any $\hookrightarrow$-pairs mentioning it. Note further that rule BOUT deals only with the export of names of entities resident in the component. The premise $\Delta; E_\Delta \nvdash T : [\![\dots]\!]$ takes care of that.[14]

Reception of new names is treated in the dual rule BIN. The component receives not only a fresh name n with the corresponding type information, but also its connectivity E'_Δ; both pieces of information are added to the respective assumption context, where it is checked that E_Δ is extended conservatively.

The last two rules deal with lazy instantiation across the component boundary. In rule BOUT_{new}, this means the reference to an object from an environment class (stipulated by $\Delta \vdash c : [\![\dots]\!]$) is communicated boundedly to the environment. Unlike as in rule BOUT, the reference is kept after scope extrusion in the *assumption* environment, as the object itself resides in the environment. The code for instantiation on part of the creator in the component must have been executed some time before by rule NEWO_{lazy} from above, which lead to a pair $o' \hookrightarrow o$ running in parallel in the component, if o' is the creator. For the transmitted connectivity in the label, it is clear that after creation of the new name o at the component side, the name is known by the creator, as indicated by $o' \hookrightarrow o$, which might give the knowledge away, of course, so that o is known to other objects from the component. The object o itself, however, does not itself know of any other object —how should it, it is not even instantiated yet.

Let us illustrate the semantics on a small example. It is a bit more complex than the similar Example 3, i.e., populated with more objects, and uses the alternative rules of this section.

Example 4 (Lazy instantiation (1)). Given two external classes c_2 and c_3 from Δ. Let further o_1 be a component object already known to the environment, i.e., $\Theta \vdash o_1 : c_1$ for some class c_1 in Θ. The following steps describe a situation, where a thread creates an object of the external class c_2, and calls a method of this object. Furthermore, it passes to the callee o_2 the reference to another object o_3, which likewise is an instance of an external class, namely c_3 and which is unknown to the environment yet. A fourth object o_4, in contrast, is contained in the component and already known to the environment, i.e., $o_4 \in dom(\Theta)$, and furthermore, o_5 is a yet unknown instance in C':

$$
\begin{array}{ll}
1 & \Delta; E_\Delta \vdash \nu(o_3{:}c_3).(C' \parallel n\langle \mathit{let}\ x{:}c' = [o_1]\ \mathit{new}\ c'\ \mathit{in}[o_1]\ x.l(o_3, o_4); t\rangle) : \Theta; E_\Theta \rightsquigarrow \\[4pt]
2 & \Delta; E_\Delta \vdash \nu(o_3{:}c_3, o_2{:}c_2).(C' \parallel o_1 \hookrightarrow o_2 \parallel n\langle \mathit{let}\ x{:}c' = o_2\ \mathit{in}[o_1]\ x.l(o_3, o_4); t\rangle) : \Theta; E_\Theta \rightsquigarrow \\[4pt]
3 & \Delta; E_\Delta \vdash \nu(o_3{:}c_3, o_2{:}c_2).(C' \parallel o_1 \hookrightarrow o_2 \parallel n\langle [o_1]\ o_2.l(o_3); t\rangle) : \Theta; E_\Theta \ \dfrac{\nu(o_2{:}c_2, o_3{:}c_3; E).n\langle [o_1]\mathit{call}\ o_2.l(o_3, o_4)\rangle !}{} \\[4pt]
4 & \Delta, o_2{:}c_2, o_3{:}c_3; E'_\Delta \vdash C'' \parallel n\langle [o_1]\ \mathit{blocks\ for}\ o_2; t\rangle : \Theta; E_\Theta
\end{array}
$$

After some internal steps, which amongst other things, create the pair $o_1 \hookrightarrow o_2$ inside the component, both the newly created o_2 and the previously created o_3 escape to the environment by lazy instantion (rule NEWO_{lazy}).

The interesting part of the reduction is the extenal step from line (3) to (4). The communicated relation E Let us abbreviate the pre-configuration of the

---

[14] Export of names by lazy instantiation is treated in rule BOUT_{lazy}.

step $\Delta; E_\Delta \vdash \nu(o_3{:}c_3, o_2{:}c_2).(o_1 \hookrightarrow o_2 \parallel n\langle[o_1] \, o_2.l(o_3); t\rangle) : \Theta; E_\Theta$ as $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$. The situation as given in line (3) is graphically sketched in Figure 2. The arrows between represent the $\hookrightarrow$-relation, the dotted arrows from c_2 and
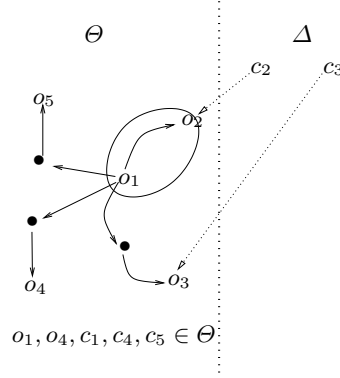

Fig. 2. Lazy instantiation

c_3 indicate that o_2 and o_3 are references to not-yet-existing instances, and the bubble around o_1 and o_2 indicates (informally) the current scope of the reference o_2; to keep the picture simple, not other scopes are represented. The $\bullet$'s are object references whose exact identity is unimportant for the example.

$$\frac{o_2 \in \Delta_2 \quad \acute{E}_\Delta^3 = E_\Delta + (o_2 \hookrightarrow (o_3, o_4, o_5), n \hookrightarrow o_2) \quad \acute{E}_\Theta^3 = E_\Theta^3 \setminus n}{\Delta^2; E_\Delta \vdash C^3 : \Theta^3; E_\Theta^3 \xrightarrow{n\langle[o_1]call\ o_2.l(o_3,o_4,o_5)\rangle!} \Delta^2; \acute{E}_\Delta^3 \vdash \acute{C}^3 : \acute{\Theta}^3; \acute{E}_\Theta^3} \text{CallOut}$$

$$\frac{}{\Delta^1, o_2{:}c_2; E_\Delta \vdash C^2 : \Theta; E_\Theta \xrightarrow{\nu(o_5:c_5;E'_\Theta)\gamma!} \Delta^2; \acute{E}_\Delta^3 \vdash C : \acute{\Theta}^3; \acute{E}_\Theta^3} \text{BOut}$$

$$\frac{}{\Delta, o_3{:}c_3; E_\Delta \vdash C^1 : \Theta; E_\Theta \xrightarrow{\nu(o_2:c_2)\gamma!} \Delta^2; \acute{E}_\Delta^3 \vdash C : \acute{\Theta}^3; \acute{E}_\Theta^3} \text{BOut}_{new}^1$$

$$\frac{}{\Delta; E_\Delta \vdash C^0 : \Theta; E_\Theta \xrightarrow{\nu(o_3:c_3).\gamma!} \Delta^3; \acute{E}_\Delta^3 \vdash C : \acute{\Theta}^3; \acute{E}_\Theta^3} \text{BOut}_{new}^0$$

	Δ	E_Δ	Θ	E_Θ	$\acute{\Delta}$	$\acute{E}_\Delta$		$\acute{\Theta}$	$\acute{E}_\Theta$
3	Δ^2	E_Δ	$\Theta, o_5{:}c_5$	$E_\Theta + E'_\Theta$	Δ^2	$E_\Delta + (o_2 \hookrightarrow o_3, o_4, o_5, n \hookrightarrow o_2)$		$\Theta^3 \setminus n$	$E_\Theta^3 \setminus n$
2	$\Delta^1, o_2{:}c_2$	E_Δ	Θ	E_Θ	Δ^2	$\acute{E}_\Delta^3$		$\acute{\Theta}^3$	$\acute{E}_\Theta^3$
1	$\Delta, o_3{:}c_3$	E_Δ	Θ	E_Θ	Δ^2	$\acute{E}_\Delta^3$		$\acute{\Theta}^3$	$\acute{E}_\Theta^3$
0	Δ	E_Δ	Θ	E_Θ	Δ^2	$\acute{E}_\Delta^3$		$\acute{\Theta}^3$	$\acute{E}_\Theta^3$

Table 14. Contexts $(o_1, o_4, c_1, c_4, c_5, n \in \Theta)$

The contexts are summarized in Table 14, where the numbers in the first column refer to the lines of the judgements of the derivation tree, and where 0 is the root at the bottom. Furthermore, by convention, the entities named $\acute{\Delta}$, $\acute{\Theta}$, etc., refer to their state after the corresponding step. Separately, Table 15 contains the form of the components; the components are read up-to $\equiv$-congruence. The primed object references are the $\bullet$'s from the pictorial representation. We assume that they are all scoped within the component, i.e., $\vec{o}':\vec{c}'$ in C^3 is the binding occurrence of the primed references. The rest C' of the component contains the actual named entitities, i.e., the primed objects just mentioned and besides that the objects and classes of the commitment context $\Theta, o_5:c_5$.

	C	$\acute{C}$
3	$o_1 \hookrightarrow o_2 \parallel n\langle[o_1]\ o_2.l(o_3,o_4,o_5); t\rangle \parallel \nu(\vec{o}{:}\vec{c}) \begin{pmatrix} o_1 \hookrightarrow o' \hookrightarrow o_4 \parallel \\ o_1 \hookrightarrow o'' \hookrightarrow o_5 \parallel \\ o_1 \hookrightarrow o''' \hookrightarrow o_3 \parallel C' \end{pmatrix}$	
2	$\nu(o_5{:}c_5)\ C^3$	$\acute{C}^3 \setminus o_5$
1	$\nu(o_2{:}c_2)\ C^2$	
0	$\nu(o_3{:}c_3)\ C^1$	

Table 15. Components

The component $\acute{C}^3$ equals C^3 where the thread n goes into the blocked state and waiting for the return of the method, as indicated in the reduction steps at the beginning of this example. The component $\acute{C}^3$ equals C^3 with all all "public acquaintances" of o_5 removed. The interconnection structure of the objects after the step is shown in Figure 3.
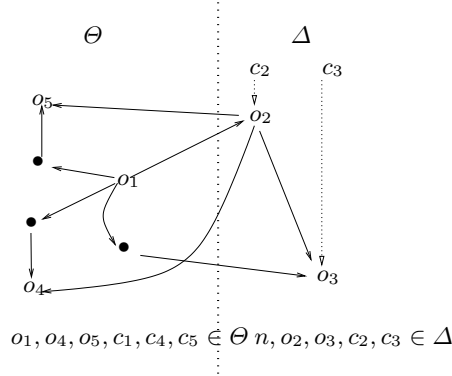

$o_1, o_4, o_5, c_1, c_4, c_5 \in \Theta\ n, o_2, o_3, c_2, c_3 \in \Delta$

Fig. 3. After the communication

In the BOUT_{new}^0-step, the context E'_Δ in the label carries the acquaintances of object o_3. The object is freshly created in this step, as is o_1, therefore there are *no acquaintances* of the form $o_3 \hookrightarrow o$, independant of whether o is a component or environment object. Of course conversely, the reference o_3 itself might be known already even if the instance is not actually created, i.e., there might be pairs $o' \hookrightarrow o_3$ inside the component, where o' is either bound in Θ or occurs (modulo structural congruence) together with $o' \hookrightarrow o_3$ in the scope of a $\nu o':c'$-binder. Note that it is *not* possible —already for reason of proper scoping— that $o' \in dom(\Theta)$ and $o' \hookrightarrow o_3 \in E_\Theta$!

The same remarks apply to o_1, the callee of the transition, whose scope is exported in the derivation step BOUT_{new}^1. Note that especially there are o_1 and o_3 cannot know of each other.

The next deriviation step —working bottom up in a goal directed manner— is an instance of BOUT justifies a "standard" scope extrusion, where only the name, namely the reference o_5 to a component instance, is exported, without any instantiation involved. A difference from the lazy instantiation steps is, that now the label carries a nontrivial connectivity part, E'_Θ in the derivation. According to the definition of rule BOUT from Table 13, E'_Θ contains all objects from the current name assumption and commitment contexts, thus in the example from

$$E_\Theta, E_\Delta, o_1{:}c_1, o_3{:}c_3 \ .$$

If we (informally) assume that the picture of Figure 2 is complete in the sense that no object and acquaintances are presents other than those indicated by the arrows and that furthermore the $\bullet$-references corresponds to component-local instances, then we can conclude:

$$E'_\Theta = o_5 \hookrightarrow o_4, o_5 \hookrightarrow o_1, o_5 \hookrightarrow o_2, o_5 \hookrightarrow o_3 \subseteq \Theta, \Delta^2 \ .$$

This exported connectivity, however, merits a closer look. First note that clearly the primed object references o', o'', and o''' are not mentioned in the relation E'_Θ. The reason is that they are not (yet) known to the environment; hence it makes no sense to commit oneself about its connectivity.[15]

More interesting is the justification of, for instance, $o_5 \hookrightarrow o_4$. In particular, the justification for this pair in E'_Θ is *not* that $\Theta; E_\Theta \vdash o_5 \leftrightharpoons\hookrightarrow o_4 : \Delta^2$! Indeed, since we assume that the intermediate names o' and o'' are not advertised via Θ to the environment but locally scoped, $\Theta; E_\Theta \nvdash o_5 \leftrightharpoons\hookrightarrow o_4 : \Delta^2$. The reason why $o_5 \hookrightarrow o_4$ must be listed on the transition label and added to E_Θ as mandated by rule BOUT is, that C^2 is $\equiv$-congruent to a component where o_5 and o_4 are connected using the reflexive, symmetric, and transitive closure from the $\hookrightarrow$-arrows in the component *as well as* the assumptions from E_Θ.[16] The same arguments justify $o_5 \hookrightarrow o_1$ as part of Θ', and similarly $o_5 \hookrightarrow o_3$ and

---

[15] More technically, if for instance o' is locally scoped within the component, there is no way to refer to it outside, for instance in the connectivity context; that's the fundamental meaning of scope.

[16] In the particular example here, E_Θ is actually not needed.

$o_5 \hookrightarrow o_2$, where the latter two are different insofar, as o_2, o_3 are members of the environment assumption Δ^2.

As mentioned, in the example, we have chosen o_1 and o_4 to be an externally known internal instances, while the argument o_3 and the callee o_2 are not yet published and exported external instances. Note that the rules does not allow an analogous step in the situation, where also the caller o_1 is an instance of an external class. □

Index