Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes

Draft technical report, July 1, 2004

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

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

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

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

1 Introduction

The notion of component is well-advertised as structuring concept for software development. At the bottom line, a component means a "program fragment" being composed, which raises the question what the semantics of a component is. A natural approach is to take an observational point of view: two components are observably equivalent, when no observing context can tell them apart.

In the context of concurrent, *object-based* programs and starting from may-testing as a very simple notion of observation, Jeffrey and Rathke [JR02] provide a fully abstract trace semantics for the language. Their result roughly states that, given a component as a set of objects and threads, the fully abstract semantics consists of the set of traces at the boundary of the component, where the traces record incoming and outgoing calls and returns. At this level, the result is as one would expect, since intuitively in the chosen setting, the only possible way to observe something about a set of objects and threads is by exchanging messages. 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 far from trivial.

The result in [JR02] is developed within the concurrent object-calculus [GH98], an extension of the sequential ν-calculus [PS93] which stands in the tradition of various object calculi [AC96] and also of the π-calculus [MPW92,SW01]. 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.[5] This is in contrast to the mainstream of object-oriented languages where the 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 threads, which are the active entities, *classes* come into play. Classes serve as a blueprint for their instances and can be conceptually understood as particular objects supporting just a method which allows to generate instances.

Important in our context is that now the division between the program fragment under observation and its environment also separates *classes:* There are classes internal to the component and those belonging to the environment. As a consequence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well. This possibility of *cross-border instantiation* is absent in the object-based setting: Objects are created by directly providing the code of their implementation, not referring to the name of a class, which means that the component creates only component-objects and dually the environment only environment objects. To understand the bearing of this change on the semantics, we must realize that the interesting part of the problem is not so much to just cover the possible behavior at the

[5] 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.

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 an obvious example, a trace with two consecutive calls from the same thread without outgoing communication in between cannot be part of the component behavior.

Let's 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 thus be traced. In other words, instances of environment classes belong to the environment, and 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.

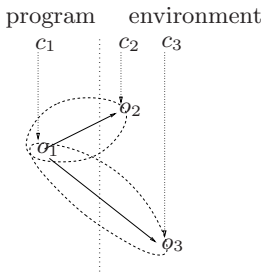


Fig. 1. Instances of external classes

In this situation it is impossible, that there be an incoming call from the environment carrying both names o_2 and o_3, as the only entity 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 references in the communication labels, the component must 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.

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 from [GH98,JR02].

Compared to the object-based 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. Since in the class-based setting we do not need general method update, we distinguish between *methods* and *fields*.

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; they may be hidden using the ν-binder but unlike object names, the scopes for class names are *static.* Object names, on the other hand, are first-order citizens of the calculus in that they can be stored in variables, passed to other objects as method parameters, making the scoping *dynamic,* and especially they can be created freshly by instantiating a class. There are no constant object names; the only way to get a new reference is instatiation.[6]

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 name n of a class serves as the type for the *named* instances of the class. Finally we need for the type system, i.e., as auxiliary type construction, the type or interface of unnamed objects, written $[l_1{:}U_1, \ldots, l_k{:}U_k]$ and the type for classes, written $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$. The grammar is shown in Table 1.

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

Table 1. Types

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

A program is given by a collection of classes, objects, and threads, where the empty collection is denoted by $\mathbf{0}$. A class $n[\![O]\!]$ carries a name n and defines the implementation of its methods and fields, whereas objects $n[n, F]$ contain only fields plus a reference to the corresponding class. A method $\varsigma(n{:}T).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t$ 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 class instantiation, creation of new threads, and a reference to the current thread. Values, finally, are either variables x or names n (and $\mathit{true}, \mathit{false}, 0, 1, \ldots$ 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.

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

Table 2. Abstract syntax

We further will use the following syntactic abbreviations and conventions. Instance variables or fields can be seen as specific form of methods, namely of empty parameter list, i.e., an instance variable declaration $f = v$ is equivalent to $f = \varsigma(n{:}T).\lambda().v$, field access $x.f$ to $x.f()$. In the class-based setting, we have to distinguish between fields, which are included into the objects and are updateable, and methods, which remain in the class, so we introduced syntactically fields as subcategory of methods. For simplicity of presentation, we adopt the convention, that when writing as class $c[\![F, M]\!]$, F contains the fields as *all* members of the required form, and the proper methods M none. It would

be straightforward, to generalize this scheme, i.e., to declare syntactically some zero-parameter members as fields (which are included into the objects in may be updated later) and other as proper methods, which remain in the classes.

The sequential composition $t_1; t_2$ of two threads stands for $\mathit{let}\, x{:}T = t_1\ \mathit{in}\ t_2$, where x does not occur free in t_2. We additionally disallow (read and write) references to fields across object boundaries.[7]

3 Type system

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

Table 3, 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 (cf. the rules T-Nu$_i$ and T-Nu$_e$). The two variants of the rule distinguish whether the bound name n, resp. o, is an instance of an external class. If not, which corresponds to rule T-Nu$_i$, the ν-bound name in $\nu(n{:}T).C$ is *not* added to the assumption context Δ in the premise, but to the commitment Θ. This combines the situations where the name stands for a instance of an internal class, or to a thread name. In this situation, the ν-construct does not only introduce a local scope for its bound name but asserts something 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. Rule T-Nu$_e$ captures the situation, when the name stands for an instance of an *external class.* In this case, the assumption context is extended, with the intuition that, once instantiated,

[7] [JR02] are slightly more general in this respect: They only forbid 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.

the object named o will reside in the environment. We call the fact that object references of external objects can be introduced and instantiated only later when first used, lazy instantiation; see Section 4 for the operational behavior.

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. again 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).

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

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

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

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

Table 3. Static semantics (components)

The typing rules of Table 4 formalize typing judgments for threads and objects and their syntactic sub-constituents. Besides assumptions about the provided names of the environment kept in Δ as before, the typing is done relative to assumptions about 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 from [JR02]. Different from the object-based setting are the ones dealing with objects and classes. Rule T-Class is the introduction rule for class types, the rule of instantiation of a class T-NewC requires reference to a class-typed name. Note also that the deadlocking expression *stop* has every type.

Later, the abbreviation $o :: T$ will be useful to express that an object reference o is an instance of a class with type T when the actual name of the class is not needed. More precisely $\Delta \vdash o :: T$ stands for $\Delta \vdash o : c$ and $\Delta \vdash c : T$. Analogously for $\Gamma; \Delta \vdash o :: T$, of course.

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

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

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

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

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

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

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

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

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

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

Table 4. Static semantics (2)

4 Operational semantics

Next we present the *operational* semantics of the calculus. The formalization is similar to the one for the object calculus, except the parts dealing with classes and especially cross-border instantiation. The basic steps of the semantics are given in two levels: internal steps, i.e., those whose effect is completely confined within a configuration, and those with external effect.

4.1 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 (cf. 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).

$n\langle let\, x{:}T = v\, in\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle$	RED
$n\langle let\, x_2{:}T_2 = (let\, x_1{:}T_1 = e_1\, in\, e)\, in\, t\rangle \rightsquigarrow n\langle let\, x_1{:}T_1 = e_1\, in\, (let\, x_2{:}T_2 = e\, in\, t)\rangle$	LET
$n\langle let\, x{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, in\, t\rangle \rightsquigarrow n\langle let\, x{:}T = e_1\, in\, t\rangle$	COND_1
$n\langle let\, x{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, in\, t\rangle \rightsquigarrow n\langle let\, x{:}T = e_2\, in\, t\rangle$	COND_2
$n\langle let\, x{:}T = stop\, in\, t\rangle \rightsquigarrow n\langle stop\rangle$	STOP
$n\langle let\, x{:}T = currentthread\, in\, t\rangle \rightsquigarrow n\langle let\, x{:}T = n\, in\, t\rangle$	CURRENTTHREAD
$c[\![F, M]\!] \parallel n\langle let\, x{:}c = new\, c\, in\, t\rangle \rightsquigarrow c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F] \parallel n\langle let\, x{:}c = o\, in\, t\rangle)$	NEWO_i
$n_1\langle let\, x{:}T = new\langle t\rangle\, in\, t_1\rangle \rightsquigarrow \nu(n_2{:}T).(n_1\langle let\, x{:}T = n_2\, in\, t_1\rangle \parallel n_2\langle t\rangle)$	NEWT
$c[\![F, M]\!] \parallel o[c, F] \parallel n\langle let\, x{:}T = o.l(\vec{v})\, in\, t\rangle \xrightarrow{\tau} c[\![M]\!] \parallel o[c, F] \parallel n\langle let\, x{:}T = M.l(o)(\vec{v})\, in\, t\rangle$	CALL_i
$o[c, F] \parallel n\langle let\, x{:}T = o.f := v\, in\, t\rangle \xrightarrow{\tau} o[c, F.f := v] \parallel n\langle let\, x{:}T = o\, in\, t\rangle$	FUPDATE

Table 5. Internal steps

The step NEWO_i describes the creation of an instance of a component *internal* class $c[\![F, M]\!]$, i.e., a class whose name is contained in the configuration. Note that instantiation is a confluent step. The fields F of the class are taken as template for the created object, and the identity of the object is new and local —for the time being— to the instantiating thread; the new named object and the thread are thus enclosed in a ν-binding. Rule CALL_i

treats an internal method call, i.e., a call to an object contained in the configuration. In the step, $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when method suite $[M]$ equals $[\dots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \dots]$. Note also that the step is a $\xrightarrow{\tau}$-step, not a confluent one. The same holds for field update in rule FUPDATE, where $[c, (l_1 = f_1, \dots, l_k = f_k, f = v').f := v]$ stands for $[c, l_1 = f_1, \dots, l_k = f_k, f = v]$. Note further that instances of a component class invariantly belong to the component and not to 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 where in the fourth axiom, n does not occur free in C_1, and the relation is imported into the reduction relations in Table 7. Note that 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. We call such traces *legal.* The legality of a trace has various aspects, where one can distinguish local and global aspects: local conditions pertain to a *single* communication label of a trace, while a global one refers to whole traces. More concretely, the only global condition will assure that a trace is well-balanced wrt. the calls and returns of each thread; this is in analogy to the treatment in [JR02] and we will come to this point later in Section 4.5.

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 $\Delta \vdash C : \Theta$. Again this general starting point is similar to the situation for the object calculus in [JR02].

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 $\textsc{NewO}_i$, 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.[8]

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

[8] 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.

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).

As a consequence must *keep track* of which identities it gives away to which object in order to exclude situations as just described. Thus the communication with the environment must be labeled appropriately, which means that transitions are labeled with the kind of communication, the thread identifier, the transmitted values, and in case of calls, the name of the method. Note that object creation is not included by a specific communication label and that the caller is not part of the label. The labels are shown in Table 8.

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

Table 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.

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 component into one clique of objects but later returns to the component coming from a different clique without connection to the first, we remember for each thread that has left the component the object from Δ it has left into.

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 with 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 (cf. Section 6.1).

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

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

Note that we close the part of $\hookrightarrow$ concerning only environment objects, 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 $\leftrightharpoons\!\!\hookrightarrow$ (in the definition, ";" denotes relational composition). 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 that 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 mentioned, besides connections between objects, E_Δ contains also information about thread names. The stored information about threads is rather restricted, though. In case the active thread has left the component, the only thing 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. See also Lemma 7 in the appendix. □

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 two 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.

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. It is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments and shown in Table 10.

Object creation across the component boundary is not immediately visible. The reason is that without constructor methods, instantiation alone and the fact that an object exists cannot be used by an observer. The only way to do observations is by method calls. Therefore, objects are incorporated only at the point when they are first communicated to the other side or used from the other side.

In [ÁBdBS03], we used as starting point a formulation of the semantics, where instantiation is recorded as an explicit step (but hidden afterwards in the traces), which we called semantics with *eager instantiation* or in short early semantics. Both formulations are equivalent. See [ÁBdBS03] for details.

To formulate the external communication, we need to augment the syntax by two additional expressions o_1 *blocks for* o_2 and o_2 *return to* o_1 v. The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1. The corresponding typing rules are shown in Table 9. Note that the return-expression has an arbitrary type, which reflects the fact that control flow never reaches the point after the return.

$$\frac{}{\Gamma; \Delta \vdash o_1 \ \mathit{blocks\ for}\ o_2 : T}\ \text{T-Block} \qquad \frac{\Gamma; \Delta \vdash v : T}{\Gamma; \Delta \vdash o_2 \ \mathit{return\ to}\ o_1\ v : T'}\ \text{T-Return}$$

Table 9. Static semantics (3)

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

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle)\quad static(\Delta, \Theta) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2)\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; \odot \hookrightarrow (\Delta', \Theta') \setminus n \\ ; \acute{\Theta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!]\quad \Delta' \vdash n : thread\quad \Theta' \vdash n : thread\quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons \odot \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta}\qquad \Delta \vdash \odot \end{array}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ return\ to\ \odot\ x\rangle : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{CalliI}_i$$

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

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2)\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \\ ; \acute{\Theta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!]\quad \Delta' \vdash n : thread\quad \Theta' \vdash n : thread\quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta}\qquad \Delta \vdash o_1 : c_1 \end{array}}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ return\ to\ o_1\ x\rangle : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{CalliI}_1$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2)\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \\ ; \acute{\Theta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!]\qquad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T}\qquad \Delta \vdash o_1 : c_1 \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta}\quad t_{blocked} = let\ x'{:}T' = o_2'\ blocks\ for\ o_1'\ in\ t \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle t_{blocked}\rangle : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel n\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ return\ to\ o_1\ x; t_{blocked}\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CalliI}_2$$

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

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

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

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

Table 10. External steps

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. The case CALLI$_i$ can be seen as special case of CALLI$_1$ and takes care of the situation where a thread crosses the border between component and environment for the first time, namely by an incoming call.

When the activity of a thread returns to the environment (cf. rule RETO), the return-statement is "popped-off" the thread; in combination with the rules for incoming calls we see that the remaining part of the thread remains blocked. Note further in this context, that the let-bound variable x in rule RETO does not occur free in the remainder of the thread t.

Calling an external object leaves the local execution in a blocked state, waiting for the matching return carrying the returned value (cf. rules CALLO and RETI). Note that the name context $\acute{\Delta}$ 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.

Incoming calls update the commitment context E_Θ in that it remembers that the callee o_2 now knows (or rather may know) the arguments $\vec{v}$, and furthermore that the thread n has entered o_2. For the role of the caller o_1 resp. $\odot$, a few more words are in order. First of all, the caller is *not* transmitted in the label which means that it remains anonymous to the callee.[9] To gauge, however, whether the call is possible in the external semantics and to adjust the bookkeeping about the connectivity appropriately, in particular when returning later, the transition chooses among possible sources of the call. Indeed, the actual identity of the caller is not needed; it suffices to know the *clique* of the caller. As representative for the clique, an equivalence class of object identities, we simply pick one object. With the only exception of the initial (external) step, the scope of at least *one* object of the calling clique must have escaped to the component, for otherwise there would be now way of the caller to address o_2 as callee. In other words, for at least one object o_1 from the clique of the actual caller (who remains anonymous), the judgment $\Delta \vdash o_1{:}c$ holds prior to the call.

The antecedent of the call-rules requires, that the caller o_1 is acquainted with the callee o_2 and with all of the arguments. In case of the very first call, we take $\odot$ as the source of the call, which is assumed to be resident in the environment. 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

[9] Of course, the caller may transmit its identity to the callee as part of the arguments, but this nevertheless does not reveal to the callee who "actually" called.

the last time: $\acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 : \acute{\Theta}$.[10] Once chosen, the assumed identity of the caller is remembered as part of the return-syntax.

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

The labels carries furthermore information as to which names are transmitted bound. For incoming calls, the binding part is of the form (Δ', Θ') where we mean by convention, that Δ' are the names being added to Δ, and analogously for Θ' and Θ. For object names, the distinction is based on the class types. For thread names, the reference is contained both in Δ' and Θ', and class names are never transmitted. For the object names in the incoming communication Δ' contains the external references which are freshly introduced to the component by scope extrusion. Θ' on the other hand are the objects which are *lazily instantiated* as part of this step inside the component. Note that while the acquaintance of the caller with the arguments transmitted free is checked against the current assumption, acquaintance with the ones transmitted bound is added to the assumption context.

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 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 clique of the caller. (cf. rule CallO).

Outgoing returns (cf. rule RetO) work analogously, except that no transmission of object identities is needed.

Remains the rule NewO$_{lazy}$ for instantiating an external class. Instead of exporting the newly created name of the object plus the object itself immediately to the environment, the name is kept local until, if ever, it gets into contact with the environment. When this happens, the new instance will not only become known to the environment, but the object will also be instantiated in the environment.[11] The actual instantiation —incoming as well as outgoing— is done when handling the exchange of bound names in the next section.

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

4.5 Trace semantics and ordering on traces

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

Not surprisingly, a major complication now concerns the connectivity of objects. In this context, the caller identity, while not visible by the callee, plays a crucial role in keeping track of assumed connectivity, in particular to connect the effect of an return to a possible caller clique. To this end, the operational semantics hypothesizes about the originator of incoming calls and remembers the guess as "auxiliary" annotation in the code for return (cf. the L-CALLI-rules from Table 10).[12]

The (hypothetical) connectivity of the environment influences what is observable. Very abstractly, the fact the observer falls into a number of independent cliques increases the "uncertainty of observation". We can point to two reasons responsible for this effect. One is that separate observer cliques cannot determine the relative order of events concerning only one of the environment cliques. To put it differently: a clique of objects can only observe the order of event projected to its own members. We will worry about this later when describing the all possible reorderings or interleavings of a given trace. Secondingly, separate observers cannot cooperate to *compare identities.* This means, as long as not in contact, the observers cannot find out whether identities sent to each of them separately are the same or not. In terms of projections to the observing clique it means that local projections are considered up to α-conversion, only.

The above discussion should not mislead us to consider the behavior of two observerving cliques as completely independent. For instance, observers can merge which means that identities, separate and local prior to the merge, become comparable and the now joint clique can find out whether local interaction of the past used the same identities or not. The absolute order of local events of the past, however, cannot be reconstructed after merging.

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

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

[12] We will us an analogous trick when describing the set of all possible behaviors under a given assumption and commitment context independant of the code in Section 6.1.

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

To sum up the discussion and independent of the details: to capture the observable behavior appropriately, we need to be able to define the projection of the external steps to the observer cliques. Now the labels for method calls in the external semantics do not contain information concerning the caller, which means given trace as a sequence of labels, we have no indication for incoming calls concerning the originating environment clique.[13]

A way to remedy this lack of information is to augment the labels as recorded in the traces by the missing information. So instead of the call label from Table 8, we use

$$n\langle [o_1] call\ o_2.l(\vec{v})\rangle \tag{3}$$

as annotated call label, where $[o_1]$ denotes the caller, respectively the clique of the caller. The augmented transition are generated simply by using the caller rules from Table 10 where the caller is added to the transition labels in the obvious way.

Remark 2 (Caller identity). An alternative formulation of the labelled transitions is possible, where there the caller is incorporated in the labels from the start. The semantics in [ÁBdBS03] was formulated that way. Since ultimately, the caller remains unobservable and is needed solely to determine a possible connectivity structure of the environment, we decided not to mention the caller identity in the labels to stress the anonymicity of the caller.

A trace of a well-typed component is a sequence of external steps. The corresponding rules are given in Table 11. For $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \stackrel{\epsilon}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$, we write shorter $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \Longrightarrow \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}$.

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

$$\frac{C_1 \Longrightarrow C_2}{\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \overset{\epsilon}{\Longrightarrow} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta}\ \textsc{Internal}$$

$$\frac{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \overset{a}{\longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}}{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \overset{a}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}}\ \textsc{Base}$$

$$\frac{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \overset{s_1}{\Longrightarrow} \Delta_2; E_{\Delta_2} \vdash C_2 : \Theta_2; E_{\Theta_2} \overset{s_2}{\Longrightarrow} \Delta_3; E_{\Delta_3} \vdash C_3 : \Theta_3; E_{\Theta_3}}{\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1} \overset{s_1 s_2}{\Longrightarrow} \Delta_3; E_{\Delta_3} \vdash C_3 : \Theta_3; E_{\Theta_3}}\ \textsc{Conc}$$

Table 11. Traces

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

The definition of projection of an (augmented) trace onto a clique of environment objects is straightforward, one simply jettisons all actions not belonging to that clique. One only has to be careful dealing with exchange of bound names, i.e., scope extrusion. The trace to start from is global and thus scope extruction of fresh names to the environment is accounted for only on a global label, namely whether the outside in its entirety has been told the name or not. From the local perspective of one environment clique, a name which has been given before by the component to another clique, nevertheless is as good as new, since it has no way of comparing the name with the identity previously sent to other cliques.

Given a global trace, its projection onto one particular clique of objects as given at the end of the trace can be defined straightforwardly by induction on the length of the trace, appending actions at the end.

Definition 2 (Projection). *Assume as trace* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \overset{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$ *and let* $\acute{\Delta}$ *contain at least one object reference, then the projection of* s *onto a clique* $[o]$ *of environment objects according to* $\acute{\Delta}; \acute{E}_\Delta$ *is written as* $s \downarrow_{[o]}$ *and defined by induction on the length of* s*:* $s \downarrow_{[o]}$ *is defined as the first component of* $(s, \Phi) \downarrow_{[o]}$*, where* $\Phi = \Delta, \Theta$*, and the projection of* $(s, \Phi) \downarrow_{[o]}$ *is given by Table 12.*

Based on the above discussion, we can now define the order on traces as follows.

Definition 3 ($\sqsubseteq_{trace}$)**.** $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$*, if the following holds. If* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \overset{sa}{\Longrightarrow} \Delta'; E'_\Delta \vdash C_1 : \Theta'; E'_\Theta$*, then* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta \overset{t}{\Longrightarrow} \Delta''; E''_\Delta \vdash C_1 : \Theta''; E''_\Theta$ *such that*

$$\frac{}{(\epsilon, \Phi) \downarrow_{[o]} = (\epsilon, \Phi)}$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad receiver(\gamma!) \notin [o]}{(t\gamma!, \Phi) \downarrow_{[o]} = (t', \Phi')}$$

$$\frac{\Phi'_2 = fn(\nu(\Phi'_1).\gamma) \setminus \Phi' \qquad (t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad receiver(\gamma!) \in [o]}{(t\ \nu(\Phi'_1).\gamma!, \Phi) \downarrow_{[o]} = (t'\ \nu(\Phi'_1, \Phi'_2).\gamma!, (\Phi', \Phi'_1, \Phi'_2))}$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad sender(\gamma?) \notin [o]}{(t\gamma?, \Phi) \downarrow_{[o]} = (t', \Phi')}$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad sender(\gamma?) \in [o]}{(t\ \nu(\Phi'').\gamma?, \Phi) \downarrow_{[o]} = (t'\ \nu(\Phi'').\gamma?, (\Phi', \Phi''))}$$

Table 12. Projection

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

If $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \Longrightarrow \Delta'; E'_\Delta \vdash C_1 : \Theta'; E'_\Theta$*, then* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta \Longrightarrow \Delta''; E''_\Delta \vdash C_1 : \Theta''; E''_\Theta$.

5 Full abstraction

In this section we address the problem of full abstraction of the calculus. We start by explaining the notion of observation we will work with.

5.1 Notion of observation

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

As starting point we choose, as [JR02], a (standard) notion of semantic equivalence or rather semantic implication —one program allows at least the observations of the other. It is based on a particular, simple form of contextual observation: being put into a context, the component, together with the context, is able to *reach* a defined point, which is counted as the successful obervation. A context $\mathcal{C}[\_]$ is a program "with a hole". In our setting, the hole is filled with a program fragment consisting of a *component* C in the syntactical sense, i.e., consisting of the parallel composition of (named) classes, named objects, and named threads, and the context is the rest of the programs such that $\mathcal{C}[C]$ gives a well-typed *closed* program $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$, where closed means that it can be typed in the empty contexts, i.e., $\vdash C' : ()$.

An appropriate way to report success in an object-oriented language is to decree one particular method reporting success and wait to observe whether the

program together with the context reaches a point where that method is invoked. This event of success reporting or observability predicate may be called *barbing*. Technically, the definitions slightly deviates from the one used in [JR02]. For the observation, there must be some visible piece of information shared between the program and the outside world, otherwise there is nothing to observe. While [JR02] use an external *object* for this purpose, in our setting, an external class, which first must be instantiated, seems more appropriate, but the choice is not very crucial as far as the resulting theory is concerned. So assume a class c_b of type $[\![succ : () \to none]\!]$, abbreviated as barb. A component C *strongly barbs on* c_b, written $C \downarrow_{c_b}$, if

$$C \equiv \nu(\vec{n}{:}\vec{T}, b{:}c_b).C' \parallel n\langle let\, x{:}none = b.succ()\, in\, t\rangle \ , \tag{4}$$

i.e., the call to the success-method of an instance of c_b is enabled.[14] Furthermore, C *barbs on* b, written $C \Downarrow_{c_b}$, if it can reach a point which strongly barbs on c_b, i.e.,

$$C \Longrightarrow C' \Downarrow_{c_b} \ . \tag{5}$$

We can now define may testing preorder [Hen88] as in [JR02].

Definition 4 (May testing). *Assume* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta$ *and* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$. *Then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{may} C_2 : \Theta; E_\Theta$, *if*

$$(C_1 \parallel C) \Downarrow_{c_b} \quad \textit{implies} \quad (C_2 \parallel C) \Downarrow_{c_b} \tag{6}$$

for all $\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C : \Delta; E_\Delta$.

5.2 Soundness

Next we prove soundness of the semantics, i.e., that the trace semantics is not "too abstract". In the proof, as well as the one for completeness, a component is interacting with a surrounding program context, i.e., both do complementary actions. Given a trace s, the dual or *complementary* trace $\bar{s}$ equals s but with all labels $\gamma!$ dualized to $\gamma?$, and vice versa.

Complementary traces describe the situation where component and environment can act together and where the complementary communication steps cancel out into internal behavior. Not only, however, is it necessary to compose traces, but also typed components. When putting together two components, their respective domains are disjoint wrt. resident named objects and classes. This disjointness does not hold, however, for named threads, since each half of the program contains its share of the threads, with all the blocked (except one) method bodies "stacked" one upon the other with the let-construct.

Now in order to compose two components, the two "halves" of each stack must be merged ("zipped") to form one combined stack. Given two components, we write $C_1 \between C_2$ for the result of the merging. As the definition is equivalent to the one in [JR02], we elide the definition here; it is given in the appendix.

[14] The notion of barbing was first introduced for the π-calculus in [MS92].

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

Proof. Assume $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, and furthermore that we are given a component C_0 as environment for C_1 and C_2 with $\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C_0 : \Delta; E_\Delta$ such that $(C_1 \parallel C_0) \Downarrow_{c_b}$.

By definition of barbing, $(C_1 \parallel C_0) \Downarrow_{c_b}$ implies that $(C_1 \parallel C_0) \Longrightarrow C' \downarrow_{c_b}$ for some component C'. The component $C_1 \parallel C_0$ is well-typed and so Lemma 10 yields $C_1 \parallel C_0 \equiv C_1 \between C_0$. Further by the decomposition Lemma 20, C_0 and C_1 can perfom complementary traces, i.e.:

$$\Delta, c_b{:}\mathsf{barb}; E_\Delta \vdash C_1 : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \Delta', c_b{:}\mathsf{barb}; E'_\Delta \vdash C'_1 : \Theta', \Psi'; E'_\Theta$$

and

$$\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C_0 : \Delta, E_\Delta \stackrel{\bar{s}}{\Longrightarrow} \Theta', c_b{:}\mathsf{barb}; E'_\Theta \vdash C'_0 : \Delta', \Psi'; E'_\Delta \ ,$$

where $C' \equiv \nu((\Delta', \Theta', \Psi') \setminus (\Delta, \Theta)).C'_1 \between C'_0$. According to Lemma 11, $C' \downarrow_{c_b}$ implies that C'_1 or C'_0 strongly barbs on c_b, i.e., we need to distinguish, whether the component C'_1 or the observer C'_0 reports success. We start with the case that the observer does.

Case: $C'_0 \downarrow_{c_b}$
The assumption $C'_0 \downarrow_{c_b}$ means that

$$\Theta', c_b{:}\mathsf{barb}; E'_\Theta \vdash C'_0 : \Delta', \Psi'; E'_\Delta \xrightarrow{\bar{a}}$$

where the external label $\bar{a}$ is of the form

$$\nu(b{:}c_b)\ n\langle \mathit{call}\ b.\mathit{succ}()\rangle! \quad \text{or} \quad \nu(n{:}\,\mathit{thread}, b{:}c_b)\ n\langle \mathit{call}\ b.\mathit{succ}()\rangle! \ .$$

Let $[o]_i$ be the cliques of Δ', Ψ' according to E'_Δ, i.e., equivalence classes of object references from Δ', Ψ'. Let furthermore s_i denote $s \downarrow_{[o]_i}$, i.e., the projection of the global trace onto the ith environment clique. Dually for $\bar{s}$ and $\bar{s}_i$. Note that the domain of Δ' is non-empty, so there is at least one equivalence class, and thus the projections are well-defined.

Now consider the $\acute{E}_\Delta$-clique of C'_0 that reports success, i.e., that issues $\bar{a}$, which is the $\acute{E}_\Delta$-equivalence class of o as mentioned in label $\bar{a}$, and assume that s_j is the corresponding projection of the global trace s.

Define s' as the longest prefix of s, such that the last action of s' is an action belonging to the clique $[o]_j$; if there is no such action in s, s' is the empty trace. The trace $\bar{s}'$ is the dual prefix of $\bar{s}$. Since the set of traces of a component is prefix-closed, we get

$$\Delta, c_b{:}\mathsf{barb}; E_\Delta \vdash C_1 : \Theta; E_\Theta \stackrel{s'}{\Longrightarrow} \Delta', c_b{:}\mathsf{barb}; E'_\Delta \vdash C''_1 : \Theta'', \Psi''; E''_\Theta$$

and

$$\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C_0 : \Delta, E_\Delta \stackrel{\bar{s}'}{\Longrightarrow} \Theta'', c_b{:}\mathsf{barb}; E''_\Theta \vdash C''_0 : \Delta'', \Psi''; E''_\Delta$$

and furthermore that C_0'' can still report success, i.e., $\Theta'', c_b{:}\mathsf{barb}; E_\Theta'' \vdash C_0' : \Delta'', \Psi''; E_\Delta'' \xrightarrow{a}$ as next step. Since $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, there exists a trace t' such that

$$\Delta, c_b{:}\mathsf{barb}; E_\Delta \vdash C_2 : \Theta; E_\Theta \stackrel{t'}{\Longrightarrow} \Delta''', c_b{:}\mathsf{barb}; E_\Delta''' \vdash C_2''' : \Theta''', \Psi'''; E_\Theta''' \ .$$

Additionally, let $[o']_j$ be the cliques of Δ'', Ψ'' according to E_Θ''' and t'_j denote $t' \downarrow_{[o']_j}$, i.e., the projection of t to the clique $[o']_j$. Then for all cliques $[o]_j$ where $i \neq j$, there exists a clique $[o']_k$ such that $t' \downarrow_{[o]_k} = t'_k \preccurlyeq s'_i = s' \downarrow_{[o]_i}$, and furthermore there exists a clique $[o']_l$ such that $t' \downarrow_{[o']_;} = t'_l = s'_j = s' \downarrow_{[o]_j}$. By Lemma 16, the oberserver C_0 can complement the shorter traces t', as well, i.e.

$$\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C_0 : \Delta \stackrel{\bar{t}'}{\Longrightarrow} \Theta''', c_b{:}\mathsf{barb}; E_\Theta''' \vdash C_0''' : \Delta''', \Psi'''; E_\Delta''' \ ,$$

followed by the success-reporting a as next step:

$$\Theta''', c_b{:}\mathsf{barb}; E_\Theta''' \vdash C_0''' : \Delta'''; E_\Delta''' \xrightarrow{a} \ .$$

By the composition Lemma 15, C_2 together with C_0 reduces as follows

$$C_2 \parallel C_0 \Longrightarrow C'''$$

where $C''' = \nu((\Delta''', \Theta''', \Psi''') \setminus (\Delta, \Theta))\ (C_2''' \between C_0''')$. Since $C_2''' \downarrow_{c_b}$, this means by Lemma 11 that $C''' \downarrow_{c_b}$, and hence $C_2 \parallel C_0 \Downarrow_{c_b}$, as required.

Case: $C_1' \downarrow_{c_b}$
This case, where the component itself reports success, cannot occur. The reason is that the component C_1 is well-typed in a context without c_b, i.e., $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta$, which means it cannot itself instantiate an object b of class c_b. Neither is it possible that its partner C_0 transmits such an object to C_1 in the trace s,[15] as this would require that C_1 contained a method whose type would mention c_b (either as argument or as return type). □

6 Completeness

6.1 Legal traces

A crucial step for completeness is an exact characterization of all possible traces of a component, the *legal* ones. The question of legality has various aspects. First of all, the calls and returns of the thread must be "parenthetic", i.e., each return must have a matching call prior in the trace. Furthermore, we must take into account whether the thread is resident inside the component or outside. If the thread is currently active inside, it cannot at the same time issue a call from outside. Finally, as already explained in connection with the operational semantics, the component must make a worst-case assumption about the relationship

[15] Note that transmission does not imply instantiation. Objects of type c_b are lazily instantiated *external* to $C_1 \parallel C_0$.

of the external objects among each other to exclude impossible combinations of transmitted object names and threads.

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

$$\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\,\Theta; E_\Theta \ ,$$

stipulating that under the type and relational assumptions Δ and E_Δ and with the commitments Θ and E_Θ, the trace s is legal. In the judgment, r represents the past history of the trace which is consulted in the derivation to assure that calls and returns appear in a balanced manner and that the component behaves deterministically. Remember from Section 3 that $o :: T$ expresses that the object reference o is an instance of a class with type T, leaving the actual name of the class unspecified. The rules for legal traces are shown in Table 13.

Distinguishing according to the nature of the first action a of the trace, the rules check whether a is possible, i.e., whether it is well-typed and adheres to the restrictions imposed by the connectivity context. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. The rules are completely symmetric wrt. incoming and outgoing communication. As for the external steps, we write the labels $\gamma!$ in the form $\nu(\Theta, \Delta).\gamma'!$, where γ' does not contain any further binders and where by convention Θ contains the name binding for instances of component classes and Δ those for environment classes.

The empty trace is always legal. An incoming call (cf. rule L-CallI) may introduce new names in the component by scope extrusion, thus adding Δ' to the assumption context in the premise. Likewise, the assumption context for connectivity E_Δ is extended by the fact that the caller knows the newly introduced references, both the ones from the environment whose name is exported to the component as well as those from Θ' which are lazily instantiated in the component. This is abbreviated by the addition of $o_1 \hookrightarrow (\Delta', \Theta')$ to E'_Δ in the premise.

In the case of the incoming call it means that the caller must be acquainted with the callee and with all of the arguments, i.e., $\acute{\Delta}; \acute{E}_\Delta \vdash o_1 \leftrightharpoons\!\hookrightarrow o_2 : \acute{\Theta}$ and $\acute{\Delta}; \acute{E}_\Delta \vdash o_1 \leftrightharpoons\!\hookrightarrow \vec{v} : \acute{\Theta}$. Furthermore we require that it is possible that the thread originates in the caller object, postulated by $\Delta; E_\Delta \vdash n \leftrightharpoons o_1 : \acute{\Theta}$. The three mentioned conditions are condensed into the one-liner $\acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta}$.

As for the external steps of the semantics in Table 10, we need to deal with the fact that the calleer identity is not part of the label. Even if anonymous, the caller plays a role insofar as calls may only be issued from cliques in contact with the callee and with other references in the label, and furthermore it is necessary for the proper bookkeeping of the connectivity assumption that the return matching to the call returns to the clique the call came from. The the rules for call labels simply nondeterministically choose one source clique, represented here by o_1, and remembers the pick in the history r. This means, stored in r are call labels augmented with the identity of the caller, which we also used in the

definition of projection (cf. Equation 3). In the rules, we write a_o for the call label augmented by o.

Coming back to the premises of rule L-CallI: For the commitment context E_Θ, simply the new knowledge is added that from now on the callee knows the arguments and that the thread n now has entered the callee o_2; the binding for n is removed from the assumption context E_Δ consequently. Finally we need to check, whether an incoming call is possible in a next step at all, i.e., whether, given the history r, the thread n is *input enabled,* (cf. Definition 15 for the definition of enabledness).

Compared to communication via method calls, communication via return slightly differs wrt. the connectedness and enabledness provisos. For a return to be possible in a next step, there has to be a matching call in the history; this is directly expressed using the *pop*-function. Furthermore, there are *no* premises requiring well-connectedness.

Remark 3. Besides taking care of the connectivity contexts, the system for legal traces presented in Table 13 differs in another aspect from the ones of [JR02]: interpreting the rules in a goal-directed manner from conclusion to premises, it works off the trace from head to tail. This is necessary since only in this way, working from head to tail, we can take into account connectivity assumptions changed by earlier interaction for later communications. The different order will also be advantageous when it comes to definability. □

6.2 Closure wrt. prefix and input receptiveness

It is crucial for the completeness proof that the constructed environment program does not just realize the given trace, but realizes it "exactly". The intuitive reason is as follows. Given a trace s_1 of a component C_1, the fact that there exists *some* environment able to perform the dual trace $\bar{s}_1$ allows us to infer that also C_2 in this context exhibits observable success. This alone does not help, since the closed program $C_{\bar{s}_1} \parallel C_2$ may reach the success-reporting state, but we need an argument that allows to conclude that the success-state is reached in a run where C_2 performs the original trace s_1! This means, given a legal trace s, the component C_s must be programmed in such a way that, when composed to a closed program, it *forces* its partner to exhibit the required trace.

In the sequential setting, this is rather straightforward, since a component is internally *deterministic.* Nevertheless, it is not strictly true that a component realizes exactly one trace: at least the set of traces of a component is *prefix closed.* Furthermore, a trace can always be extended by an incoming communication provided the component is input enabled, or that there is a due return.

Additionally, deterministic

Definition 5 (Information preorder). *The* information preorder *on traces* $\Delta; E_\Delta \vdash s_1 \sqsubseteq s_2 : \Theta; E_\Theta$ *is the reflexive and transitive relation generated by the axioms from Table 14. The traces left and right of* $\sqsubseteq$ *are tacitly assumed to be legal.*

The two rules reflect that traces are closed under prefix and that traces can always be extended by an input. This is to be understood under the restriction that both traces are legal according to the rules from Table 13. Especially, the connectivity assumptions and commitments E_Δ and E_Θ must be met. Note further that a component is enabled not only wrt. incoming calls but also wrt. incoming returns. Also here, the legality assumption takes care that s can be extended to $s\ \nu(\Delta).n\langle return(v)\rangle?$ only if the trace is still balanced.

6.3 Definability

This section contains one the key to completeness. We start by sketching the line of argument in the next section, before we provide the definability construction in detail.

6.3.1 Outline of argument At the heart of the completeness result lies *definability*: construct a program that realizes as exact as possible a given legal trace. The construction of C_s, the component realizing a legal trace s, is therefore by induction on the derivation using the rules of Table 13. Interpreting the rules in a goal-directed manner, they check a trace for legality from head to tail, or reading the rules conversely from premise to conclusion, the trace is extended at its head. Thus the inductive definition of C_s constructs a program in a *backward manner*, starting from a program realizing the empty trace and extending it step by step with *preceding* environment interaction.

Besides the overall construction strategy, let us have a closer look what we need to realize the construction. The rule system derives legality judgments of the form

$$\Delta, E_\Delta \vdash r \rhd s : trace\, \Theta, E_\Theta\ ,$$

where, as usual, Δ and E_Δ are the assumptions about the environment, and dually Θ and E_Θ the commitments of the component to be constructed. Therefore, Θ and E_Θ pose requirements on C_s which have to be programmed.[16]

Before we embark on the inductive construction of C_s itself, it is instructive to abstractly think of which requirements the commitment assumptions express and how the legality rules manipulate them, since Θ and E_Θ must be implemented by some *"data structures"* and their changes by some *"algorithms"*.

The context Θ lists all named components which have to be present and publicly visible in C_s. As named components we have classes, objects, and threads.

[16] As an aside: We write C_s for the inductively defined component realizing s. To be overly precise, the component should be "indexed" by the *derivation* justifying legality of s. Note however that the trace s *determines* the derivation, at least what the sequence of applied rules is concerned. The rule system is not completely deterministic, however, as the connectivity of new objects can be guessed in different ways.

Concentrating for now on the objects, Θ is changed by scope extrusion when internally created objects get exposed to the environment, i.e., their scope opens across the component boundary. This sure can be the case for outgoing communication, but in the lazy instantiation scheme we employ, also incoming communication may make the component aware that the environment has created instances of internal classes.

The connectivity context $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$ stipulates for component objects from Θ, which other objects from Θ and from Δ it is expected to know and which it is able to contact, when necessary. In principle, there are various ways to implement E_Θ. We adopt a *distributed* implementation, i.e., each object contains in its instance state its share of E_Θ.[17] Being kept distributed over the members of the clique, changes to E_Θ must be propagated to all members.

We describe the implementation and the propagation not yet in concrete ν-calculus terms and postpone the problem to ultimately *encode* the solution into classes and objects (cf. Section **??**).

The fact that E_Θ is implemented in a distributed way does not literally mean that each object has a local copy of the full E_Θ available in its instance state, rather than its *local view* of E_Θ. This means each object keeps in its instance state the list of objects from Θ as well as from Δ it is aware of. In slight abuse of notation we call the respective instance variable Θ; to distinguish it from the abstract context Θ we will always reference it in a qualified manner as $self.\Theta$.[18]

As mentioned before, the distributed implementation of E_Θ makes it necessary to broadcast across the clique any change to the connectivity context. This change of E_Θ occurs in the system for legal traces when dealing with *incoming* communication —reception of names may increase the knowledge of the component clique— and for *outgoing* communication —*new* objects previously unknown to the environment are exported; their connectivity is guessed and the resulting change of E_Θ has to be implemented.

In any case, the change of knowledge takes place in *some* object, namely the caller resp. the callee in the communication. The object therefore has to update its own knowledge accordingly *and* to inform all members of its clique about the change. Other changes to Θ resp. E_Θ requires to create new, appropriately connected objects.

6.3.2 Static semantics with connectivity The judgment now is of the form

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \ . \tag{7}$$

Δ and E_Δ play again the role of assumptions; dually Θ and E_Θ are commitments for the contexts. The connectivity contexts E_Δ and E_Θ are interpreted therefore as follows. The assumption E_Δ stipulates that the connectivity of the

[17] How to actually encode it in the calculus, we will discuss later.

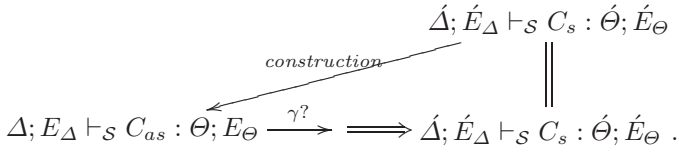
[18] Note that $self.\Theta$ corresponds more to a instance-local view of E_Θ rather than Θ, and can also contain objects from Δ.

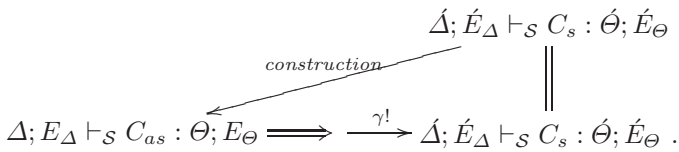
(known) objects of the environment is *at most* as given in E_Δ. Analogously, the commitment asserts that the connectivity inside the component is not higher than specified by E_Θ. This means, given the judgment $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$, all objects listed in Θ must be present in C, but for the connectivity, E_Θ means, all acquaintances of objects from C are covered by E_Θ. Especially, E_Θ makes no assertion about actually connectivity, it is rather the contrapositive statement: in terms of "commitment" or "guarantee", the context E_Θ can be thought of guaranteeing that objects are *not* connected in the component C, if they acquaintance cannot be deduced from E_Θ.

6.3.3 Inductive definition of C_s With the auxiliary definitions of the previous section at hand, we are now in a position to define the component realizing a given trace. The construction is given in the following proposition, where a number of properties of the construction are proved at the same time.

Proposition 2. *Given $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\,\Theta; E_\Theta$, the component C_s is defined by induction on the derivation of the legal-trace judgment by the cases in the proof given below. Furthermore, the constructed component C_s has the following properties.*

1. *It is well-typed, i.e., $\Delta \vdash C_s : \Theta$.*
2. *It is fully connected, i.e., $\Delta; E_\Delta \vdash_{\mathcal{S}} C_s : \Theta; E_\Theta$.*
3. *It is instance closed.*
4. *Depending on the enabledness condition of thread n after trace r, C_s is likewise enabled.*
5. *Assume $\Delta; E_\Delta \vdash r \rhd as : \mathit{trace}\,\Theta; E_\Theta$ with subgoal $\acute{\Delta}; \acute{E}_\Delta \vdash ra \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E}_\Theta$ and C_{as} given according to the construction. Then $\Delta; E_\Delta \vdash_{\mathcal{S}} C_{as} : \Theta; E_\Theta \overset{a}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C_s : \acute{\Theta}; \acute{E}_\Theta$. Pictorially:*





Proof. Given C_s, we provide the construction of C_{as} for the various rules for legal traces from Table 13. At the same time we prove preservation of the properties claimed in the proposition.

Case: L-EMPTY (base case)
The component $\Delta; E_\Delta \vdash C_\epsilon : \Theta; E_\Theta$ is of the following form. For each class name c with $\Theta \vdash c : [\![T]\!]$, it contains a class $c[\![O]\!]$ realizing all required methods in an empty way. I.e., if $[\![T]\!] = [\![\dots, l{:}U, \dots]\!]$ with $U = \vec{T} \to T$, then O contains a method labelled l and $O.l = \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).\, \mathit{stop}$. Note that the expression *stop* has any type (rule T-STOP). Likewise for all exported objects names o, i.e., for all o with $\Theta \vdash o :: T$, there exists an object $o[O]$ in the configuration realizing as in the case of classes, all the required methods in an empty way. I.e., if $[\![T]\!] = [\![\dots, \vec{T} \to T', \dots]\!]$, then O contains a method labelled l with $O.l = \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).\, \mathit{stop}$. Finally, if $\Theta \vdash n : \mathit{thread}$ for some thread name n, the component contains $n\langle \mathit{stop} \rangle$.

The component does not contain ν-binders. Especially all components mentioned above —classes, objects, and potentially one thread— are visible outside.

It is easy to check, that $\Delta; E_\Delta \vdash C_\epsilon : \Theta; E_\Theta$ is well-typed, well-connected, and instance closed. Also the enabledness requirement of part 4 is satisfied: after the empty trace, the thread is input- and output-enabled, which is covered by the forms $n\langle \mathit{stop} \rangle$ or by the situation, when there is no thread in the component.

Case: L-CALLI: $a = \nu(\Delta', \Theta').\ n\langle \mathit{call}\ o_2.l(\vec{v}) \rangle$?
The configuration $\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C_s : \acute{\Theta}; \acute{E}_\Theta$ after accepting the call, i.e., for trace s in the premise of L-CALLI, is of the form:

$$\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel \Theta'(\vec{O}) \parallel n\langle t_{ore} \rangle : \acute{\Theta}; \acute{E}_\Theta\ ,$$

with $t_{ore} = t^o_{body}; t_{ie}$. The premise $\Delta \vdash r \rhd a : \Theta$ of rule L-CALLI asserts that after r and prior to the call, the thread n is input-call enabled. By Lemma 24, n is output-return enabled after the trace ra. By assumption 4 of the proposition, C_s is output-return enabled, i.e., of the form t_{ore} from Table 16.

By the premises of the rule we know that $\acute{\Theta} = \Theta + \Theta'$, where Θ' contains (besides potentially the thread identity) the identities of the lazily instantiated component objects. The parallel component $\Theta'(\vec{O})$ abbreviates the instances of all lazily instantiated objects from Θ' in their initial state. I.e., if $\Theta' = o'_1{:}c'_1, \dots, o'_k{:}c'_k$ or $\Theta' = o'_1{:}c'_1, \dots, o'_k{:}c'_k, n{:}\,\mathit{thread}$, then $\Theta'(\vec{O})$ stands for $o'_1[O'_1] \parallel \dots \parallel o'_k[O'_k]$, where for all class names c'_i, the component $C'_s \equiv c'_i[\![O'_i]\!] \parallel C''_s$. Note that instance closedness guarantees the well-definedness of this assumption, since as a consequence C'_s contains the classes required for the lazy instantiation of the new objects.

Depending on whether the callee o_2 is lazily instantiated or not, i.e., whether $o_2 \in \Theta' \setminus \Theta$ or $o_2 \in \Theta \setminus \Theta'$, the instance o_2 is part of C'_s or of $\Theta'(\vec{O})$.

If $o_2 \in \Theta \setminus \Theta'$, the configuration after the call more specifically is of the form

$$\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C''_s \parallel o_2[c_2, \acute{F}_2] \parallel c_2[\![\acute{M}_2]\!] \parallel \Theta'(\vec{O}) \parallel n\langle t_{ore} \rangle : \acute{\Theta}; \acute{E}_\Theta\ .$$

Let $\tilde{C}_s$ denote C_s except $n\langle t_{ore}\rangle$. Now the configuration prior to the call for C_{as} is *constructed* in the following steps. Let

$$\tilde{C}_s^3 \triangleq \tilde{C}_s \oplus (c_2, \check{a}, t^o_{body}) \tag{8}$$

$$\tilde{C}_s^2 \triangleq \tilde{C}_s^3 \oplus (\vec{O}_2, \check{a}) \tag{9}$$

$$\tilde{C}_s^1 \triangleq \tilde{C}_s^2 \oplus (\Theta'(\vec{O}), \mathit{script})) \tag{10}$$

$$\tilde{C}_{as} \triangleq \tilde{C}_s^1 \setminus \Theta'(\vec{O}) \tag{11}$$

using the (overloaded) operator from Definition 11 and 12. In the second step, the set of objects $\vec{O}_2$ is defined as $\vec{O}_2 = \{o \mid \acute{\Theta}; \acute{E}_\Theta \vdash o_2 \hookrightarrow o : \acute{\Delta} \wedge \acute{\Theta} \vdash o : c\}$.[19] Note also the order of the second and the third step. In the third step, all the futures of the newly instantiated component instances (by lazy instantiation) are included as potential behavior into the respective classes. The change of the *scripts* instance variable in the classes is therefore done *after* the individual *script*-instance variabe of the objects have been changed in the second construction step. Indeed, the order of the three construction steps here is the exact reverse to the synchronization code added (cf. Definition 10). The last step, defining $\tilde{C}_s^0 = \tilde{C}_{as}$, removes the newly instantiated objects $\Theta(\vec{O}')$ from the compent, unsetting also the respective references to them. Thus, the component $\Delta; E_\Delta \vdash C_{as} : \Theta; E_\Theta$ before the call is given by:

$$\Delta; E_\Delta \vdash n\langle t_{ie}\rangle \parallel \tilde{C}_{as} : \Theta; E_\Theta \tag{12}$$

If otherwise o_2 is lazily instantiated in the call, i.e., $o_2 \in \Theta' \setminus \Theta$, the configuration for C_{as} is given as:

$$\Delta; E_\Delta \vdash C_s'' \parallel n\langle t_{ie}\rangle \ : \Theta; E_\Theta \ .$$

Similarly, if the thread identity n has entered the component in this call for the first time, i.e., if $n \in \Delta' \cap \Theta'$, then the named thread $n\langle t_{ie}\rangle$ is not present in the configuration. Note that $(n\colon \mathit{thread}) \in \Delta \cap \Theta$ iff $(n\colon \mathit{thread}) \in \Delta$ iff $(n\colon \mathit{thread}) \in \Theta$, and analogously for the context pair Δ' and Θ' transmitted in the label.

Well-typedness of $\Delta; E_\Delta \vdash C_{as} : \Theta; E_\Theta$ follows directly from the well-typedness assumption $\acute{\Delta}; \acute{E}_\Delta \vdash C_s : \acute{\Theta}; \acute{E}_\Theta$. Likewise, well-connectedness of the constructed C_{as} follows from well-connectedness of the given C_s. instantiated

By construction in rule L-CALLI, $\acute{E}_\Theta$ contains the pair $n \hookrightarrow o_2$ after the call, but the commitment context for connectivity does not contain a pair mentioning n. By one of the premises, however, we know $\acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons [o_1] : \acute{\Theta}$, which means in the case where $(n\colon \mathit{thread}) \in \Delta \cap \Theta$, that E_Δ contains a pair $n \hookrightarrow o$ for some object identifier o.

[19] This means: we add $\check{a}$ to the instance variable *script* of all acquaintances of o_2 using the connectivity *after* the step, which might be larger than the connectivity before the step, in case the step merges o_2's clique. Intuitively this reflects the fact that the merging action is counted as common action already.

As mentionend above, n is input-enabled before the call, i.e., after r. By construction, t_{ie} satisfies the condition 4 of the proposition.

Concerning the *reduction* from the pre- to the post-configuration as stated in part 5, we argue as follows:[20]

$$
\begin{array}{ll}
\Delta; E_\Delta \vdash_{\mathcal{S}} C_{as} : \Theta; E_\Theta & = \\
\Delta; E_\Delta \vdash n\langle t_{ie}\rangle \parallel \tilde{C}_{as} : \Theta; E_\Theta & \xrightarrow{a} \\
\acute{\Delta}; \acute{E}_\Delta \vdash n\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ return\ to\ o_1\ x; t_{ie}\rangle \parallel \tilde{C}^1_s : \acute{\Theta}; \acute{E}_\Theta & \xrightarrow{\tau} \\
\acute{\Delta}; \acute{E}_\Delta \vdash n\langle let\ x{:}T = M_2.l(o_2)(\vec{v})\ in\ o_2\ return\ to\ o_1\ x; t_{ie}\rangle \parallel \tilde{C}^1_s : \acute{\Theta}; \acute{E}_\Theta & = \\
\acute{\Delta}; \acute{E}_\Delta \vdash n\langle let\ x{:}T = t_{body}[o_2/s][\vec{v}/\vec{x}]\ in\ o_2\ return\ to\ o_1\ x; t_{ie}\rangle \parallel \tilde{C}^1_s : \acute{\Theta}; \acute{E}_\Theta & \Longrightarrow \\
\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} n\langle let\ x{:}T = t^a_{ore}\ in\ o_2\ return\ to\ o_1\ x; t_{ie}\rangle \parallel \tilde{C}^3_s : \acute{\Theta}; \acute{E}_\Theta & = \\
\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} n\langle t_{ore}\rangle \parallel (\tilde{C}_s \oplus (c_2, \breve{a}, t^o_{body}) : \acute{\Theta}; \acute{E}_\Theta &
\end{array}
$$

The reduction uses the rules from Section 4 and for the internal reduction especially the Lemma 3 for input call synchronization. Note that all steps are those of thread n and that only the very first step is an external one, namely a.

Case: L-RetO: $a = \nu(\Theta', \Delta').\ n\langle return(v)\rangle$!
The configuration $\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C_s : \acute{\Theta}; \acute{E}_\Theta$ after the return, i.e., for trace s in the premise of rule L-RetO, is of the form

$$\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t_{ie}\rangle : \acute{\Theta}; \acute{E}_\Theta\ .$$

By premise $pop\ n\ r = \nu(\Delta'', \Theta'').\ n\langle[o_1]call\ o_2.l(\vec{v})\rangle?$ of rule L-RetO, the thread is output-return enabled after r. By Lemma 24, n is input enabled after ra. By assumption 4 of the proposition, t_{ie} is of the form as given by Table 16.

Furthermore by the premise of the derivation rule, $(\acute{\Theta}, \acute{E}_\Theta) = (\Theta, E_\Theta) + \Theta'$, where $\acute{E}_\Theta$ is a conservative extension of E_Θ. We define the configuration prior to the return for C_{as} by

$$\Delta; E_\Delta \vdash C'_s \parallel n\langle let\ x{:}T = t^o_{sync}(\Theta', \Delta'); \check{v}\ in??\ return\ to\ o_1\ x; t_{ie}\rangle : \Theta; E_\Theta\ . \tag{13}$$

After trace r, i.e., before the outgoing return, the thread is, as said, output-return enabled and the constructed trace satisfies the condition 4 of the proposition for t_{ore}. Well-typedness and well-connectedness are checked straightforwardly.

Concerning the reduction required in part 5, the sequence follows by Lemma 1 and by rule RetO in combination with the binding rules BOut and BOut_{new}.[21]

[20] The given sequence of steps assumes that o_2 is not lazily instantiated in the call and hence already present in the component prior to it, and likewise the thread named is already present. The derivation thus corresponds to the situation of Equation (12) from above. The situations where o_2 is lazily instantiated or where n enters the component for the first time work analogously.

[21] The rule BOut_{thread} is not needed: No thread name extrudes its scope via a return, no matter whether incoming or outgoing. This is guaranteed by the requirement that the trace is balanced.

Case: L-CALLO: $a = \nu(\Theta', \Delta').\ n\langle call\ o_2.l(\vec{v})\rangle!$

The premise $\Delta \vdash r \rhd a : \Theta$ of L-CALLO stipulates that after r, i.e., prior to the call, n is output-call enabled. Hence by Lemma 24, n is input-return enabled after history ra, and thus by assumption in part 4 of the proposition, the configuration $\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C_s : \acute{\Theta}; \acute{E}_\Theta$ after the outgoing call, i.e., for the trace s in the premise of rule L-CALLO, is of the form[22]

$$\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} n\langle t^i_{body}; t_{ie}\rangle \parallel C'_s : \acute{\Theta}; \acute{E}_\Theta$$

for some component C'_s and with t^i_{body} of the form

$$let\ y{:}T' = o_1\ blocks\ for\ o_2\ in\ (let\ x{:}T = t^i_{sync}(y); t_{oe}; t^o_{sync}; v\ in\ o_2\ return\ to\ o_1\ x)$$

We distinguish whether the thread is exported for the first time to the environment.

Subcase: $\Theta \vdash n : thread$
By the premises of L-CALLO, the commitment contexts Θ and especially E_Θ are extended in the step from C_{as} to $\acute{C}_s$ in that $(\acute{\Theta}, \acute{E}_\Theta) = (\Theta, E_\Theta) + \Theta'$, where $\acute{E}_\Theta$ is a conservative extension of E_Θ. Thus the configuration for the component C_{as} is defined as:[23]

$$\Delta; E_\Delta \vdash_{\mathcal{S}} n\langle t^o_{body}; t_{ie}\rangle \parallel C'_s : \Theta; E_\Theta \tag{14}$$

with t^o_{body} given as

$$let\ x{:}T = t'_{oe}; t^o_{sync}; v\ in\ \ o_2\ return\ to\ o_1\ x\ , \tag{14}$$

which expands further to

$$let\ x{:}T = (t^o_{sync}; let\ y{:}T' = o_2.l(\vec{v})\ in\ t^i_{sync}(y); t_{oe}; t^o_{sync}; v)\ in\ \ o_2\ return\ to\ o_1\ x\ . \tag{14}$$

As mentionend above, the trace is output (call) enabled after r, and the configuration for C_{as} complies to the requirement of part 4 of the proposition. The well-typedness and well-connectedness conditions are again checked straightforwardly.

[22] We assume that n is already known in the environment. Otherwise, n would occur in C_s under a ν-binder and escape via scope extrusion in the call-step.

[23] Note that both the classes from Θ' as well as the ones from Δ' are handled by the create-statement; the latter ones are the lazily instantiated objects, which will reside in the environment after the communication.

For part 5 we obtain:

$$
\begin{array}{ll}
\Delta; E_\Delta \vdash_{\mathcal{S}} C_{as} : \Theta; E_\Theta & = \\
\Delta; E_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t^o_{body}; t_{ie}\rangle : \Theta; E_\Theta & \Longrightarrow \\
\Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\Theta', \Delta').(C'_s \parallel n\langle let\, x{:}T = (let\, y{:}T' = o_2.l(\vec{v})\ in\ t^i_{sync}(y); t_{oe}; t^o_{sync}; v)\ \ in\, o_2\ \ return\ \ to\ \ o'_1\ \ x\rangle \parallel \vec{o}[\vec{c}]) : \Theta; E_\Theta & \rightsquigarrow \\
\Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\Theta', \Delta').(C'_s \parallel n\langle let\, y{:}T' = o_2.l(\vec{v})\ in (let\, x{:}T = t^i_{sync}(y); t_{oe}; t^o_{sync}; v\ \ in\, o_2\ \ return\ \ to\ \ o'_1\ \ x)\rangle \parallel \vec{o}[\vec{c}]) : \Theta; E_\Theta & \xrightarrow{a} \\
\Delta; E_\Delta \vdash \nu(\Theta', \Delta').(C'_s \parallel n\langle let\, y{:}T' = o_1\ \ blocks\ for\ o_2\ in (let\, x{:}T = t^i_{sync}(y); t_{oe}; t^o_{sync}; v\ \ in\, o_2\ \ return\ \ to\ \ o'_1\ \ x)\rangle \parallel \vec{o}[\vec{c}]) : \Theta; E_\Theta & = \\
\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t^i_{body}; t_{ie}\rangle \parallel \vec{o}[\vec{c}] : \acute{\Theta}; \acute{E}_\Theta & = \\
\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t_{ire}\rangle \parallel \vec{o}[\vec{c}] : \acute{\Theta}; \acute{E}_\Theta &
\end{array}
$$

where $\vec{o}[\vec{c}]$ abbreviates $o_1[c_1] \parallel \ldots \parallel o_k[c_k]$ and furthermore $\acute{\Delta} = \Delta + \Delta'$, $\acute{\Theta} = \Theta + \Theta'$, and $\acute{E}_\Theta = E_\Theta + E(C, \Theta', \Delta')$.

In the reduction sequence, the crucial steps from line 2 to 3 are covered by Lemma 1.

Subcase: $\Theta' \vdash n : thread$
In this case the thread n is exported to the environment for the first time. There are further two subcases. In case $\Delta; E_\Delta \vdash n' : thread$ for some thread n', i.e., n is not the initial thread, then component contains a named thread of the form $n'\langle t\rangle$ and the component C_{as} is given by $\Delta; E_\Delta \vdash_{\mathcal{S}} \tilde{C} \parallel n'\langle t^{ont}_{sync}; t\rangle : \Theta; E_\Theta$.

For both cases, the required properties are shown as in the first subcase.

Case: L-RetI: $a = \nu(\Delta', \Theta')$. $n\langle return(v)\rangle$?
By the premise $pop\, n\ r = \nu(\Delta'', \Theta'').n\langle[o_1]call\ o_2.l(\vec{v})\rangle$! of L-RetI, n is input-return enabled after r. Furthermore, the premise reveils the identity of the caller. By Lemma 24, n is output enabled after ra. By assumption from part 4, the configuration for C_s after the incoming return, i.e., for the trace s in the premise of L-RetI, is of the form

$$\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t_{oe}\rangle : \acute{\Theta}; \acute{E}_\Theta$$

for some component C'_s and thread t_{oe} according to Table 16. We construct the component before the incoming return as follows:[24]

$$\Delta; E_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t^i_{body}; t_{ie}\rangle \tag{15}$$

with t^i_{body} equalling

$$
\begin{array}{l}
let\, y{:}T' = o_1\ \ blocks\ for\ o_2\ in \\
\qquad (let\, x{:}T = t^i_{sync}(y, \check{a}); t_{oe}; t^o_{sync}; v\ in\ \ return\ \ to\ \ \ x)
\end{array}
\tag{15}
$$

As mentioned, thread n is input-return enabled after r and the constructed C_{as} conforms to the requirements stated in 4 of the proposition.

[24] By another premise we know that $\acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons\!\hookrightarrow o_2 : \acute{\Theta}$ which means that $\acute{E}_\Delta$ must contain a pair $n \hookrightarrow o'_2$ for some object identifier o'_2. But note that the callee o_2 need not coincide with o'_2! The latter o'_2, stored in the assumption context, remembers the object the thread n has visited when it had left the component the *last time.* The *pop*-operation, on the other hand, finds the last *matching* call.

For part 5 we reason as follows: By rule RetI in combination with the scoping rules BIn and BIn_{new}, and furthermore by the Lemma 2 for input synchronization we can derive:

$$
\begin{array}{ll}
\Delta; E_\Delta \vdash_{\mathcal{S}} C_{as} : \Theta; E_\Theta & = \\
\Delta; E_\Delta \vdash_{\mathcal{S}} C'_{as} \parallel n\langle t^o_{body}; t_{ie}\rangle : \Theta; E_\Theta & \xrightarrow{a} \\
\acute{\Delta}; \acute{E}_\Delta \vdash C'_s \parallel n\langle let\ y{:}T' = v\ in\ (let\ x{:}T = t^i_{sync}(y, \check{a}); t_{oe}; t^o_{sync}; v'\ in\ \ return\ to\ \ x); t_{ie}\rangle : \acute{\Theta}; \acute{E}_\Theta & \rightsquigarrow \\
\acute{\Delta}; \acute{E}_\Delta \vdash C'_s \parallel n\langle let\ x{:}T = t^i_{sync}(v, \check{a}); t_{oe}; t^o_{sync}; v'\ in\ \ return\ to\ \ x; t_{ie}\rangle : \acute{\Theta}; \acute{E}_\Theta & \Longrightarrow \\
\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle let\ x{:}T = t_{oe}; t^o_{sync}; v'\ in\ \ return\ to\ \ x; t_{ie}\rangle : \acute{\Theta}; \acute{E}_\Theta & = \\
\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C'_s \parallel n\langle t_{ore}\rangle : \acute{\Theta}; \acute{E}_\Theta &
\end{array}
$$

□

Before we illustrate the construction on a few simple examples, we analyze the cases of the definition in more detail.

For *outgoing calls,* the message is issued from some object which is acquainted with the one mentioned in the label as sender. A further effect of an outgoing call is, that *new internal* objects are communicated to the environment; they are mentioned in Θ' and must be created, using the *create*-statement (cf. Equation 14). Besides that, new *external* object can be created in a lazy manner, in that Δ is extended to $\acute{\Delta}$, but this is under the responsibility of the environment.

For an *incoming return,* we prefix the thread by a block-expression to accept the incoming return (cf. Equation 15). The incoming information may increase the knowledge base E_Θ to $\acute{E}_\Theta$. This information has to be propagated through the component. The component can learn new names in an incoming return, but there can be no new thread name coming in. Therefore, no new threads have to be created in this case. References to external instances, as introduced by the extension from Δ to $\acute{\Delta}$, are under the responsibility of the environment. The premise $\acute{\Theta} = \Theta + \Theta'$, finally, describes the lazy cross-border instantiation of internal objects by the environment. Neither these have to be created by C_s.

Working backwards in the program construction, an *incoming call* is the place where we *"wrap up"* a method. The code of the method body must be added to the object (and the class) and bound to the method label label l (cf. Equation 12). For an *outgoing return* according to case L-RetO, finally, we again have to take care as in the case for outgoing calls, that all internal external object references exported to the outside via Θ' are created before handing back the return value (cf. Equation 13).

6.3.4 Synchronization code Next we fill in in more detail the missing pieces of code, namely the one for *synchronization* in the construction from Proposition 2. We start with an abstract description of what the synchronization is good for.

Overview The pieces of synchronization code in the backward construction of the component C_s come in two flavors, *input* and *output* synchronization code,

and flank the corresponding external transition steps at the interface. Output sychronization code *precedes* the corresponding output, and dually, input synchronization *trails* the input action (cf. also the two diagrams in the formulation of Proposition 2).

As the commitment contexts of the judgments are nothing else than a concise specification of the component, what the synchronization has to achieve can be clearly understood by looking at the change of the $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$- judgments in external steps (cf. Tables 10, resp. Table 13). The changes are always *additive,* i.e., the contexts only grow larger. To implement the extension of the typing context Θ in an output step, the component must *create* corresponding objects, whose references are then published. Likewise the component must cater for lazily instantiated objects of the environment, which lead to an extension of E_Δ in an output step. On the other hand, the component is not responsible for extensions of E_Θ by incoming lazy instantiation.

For connectivity as specified by E_Θ, we adopted a "distributed" implementation, i.e., all objects in a clique are kept fully-connected. Thus, if the connectivity context E_Θ is enlarged in an input step, the additional information must be distributed to all members of the clique.

We split the abstractly described synchronization task into the following four sub-problems:

1. to *create* new objects to be made known or exported to the outside,
2. to *propagate* connectivity information to keep the component fully connected,
3. to *serialize* the actions of the component to exhibit *exactly* the behavior prescribed by the trace,[25] and
4. to provide "mutual exclusion" to avoid interference.

The first point is straightforward: the synchronization code for output will contain appropriate *new*-statements. The second point will be done by traversing the clique, updating the connectivity knowledge of all of its members.

The serialization task mentioned in point 3 is not implied by the previous discussion about how the commitment contexts and their change specify the implementation task. It is mandated by the completeness proof in general. Anyway, the task is to ensure that the actions and reactions of the component follow the prescibed order (to the extent possible). For instance, consider that the trace contains the following sequence of two actions $\gamma?.\ \gamma'!$, say

$$n\langle \mathit{call}\ o_2.l(\vec{v})\rangle?.\ n'\langle \mathit{call}\ o_2'.l'(\vec{v}')\rangle! \ ,$$

where $n \neq n'$ and where the two actions concern the same clique. In this situation the implementation must *enforce* the given order, i.e., it is necessary to assure

[25] Of course, the component cannot completely serialize its behavior, for various reasons. Especially, separate cliques cannot enforce a particular order of their respective events.

that thread n' does not issue the second call *before* the first incoming call has been accepted.

To achieve this, each object of the clique must be aware (and kept up-to date) of the current status wrt. the sequence of interactions at the clique's interface. In the above situation for instance, the caller object of the second, outgoing call, must be aware whether or not the first call $\gamma?$ has already occurred.

In the concrete constructed component, the objects do not keep a history of past interaction. Rather the current state is characterized by the *future interaction* the component still has to realize. We call such a linear description of the future of an object a *script.*[26] The constructed component equips each object (and ultimately each class) with one possible linear behavior. Since the instances of the class may have to behave differently according to the given legal trace, the class contains a *set* of possible futures. Concretely, the future behavior is implemented by an instance variable *script* containing one sequence of *actions*, while the class collects all possible futures in the *scripts* (plural) instance variable. When an object is instantiated from a class, one future is picked at random.[27] We maintain as invariant that all objects of a clique always agree on the common future. We call such a clique *script-consistent.*

The *output* synchronization code checks, before it does the actual output, whether

1. the component is (still) script-consistent and, if so,
2. whether the intended output action is indeed "next on the list",

i.e., whether the future stored in *script* starts with the intended output action. If so, the code cancels the action from the script and broadcasts this fact throughout its clique. Before it does the check and the canceling the intended action is not the expected one, the thread *blocks.* We use a guarded command like syntax $\breve{a} \rhd t$, where $\breve{a}$ is the intended action (see below). Note that we need to check for a consistent future in an output step since the clique may be enlarged, namely by the newly created objects. Their future must be fit together with the future of the rest of the clique.

For *input* synchronization, the situation is similar: the thread is allowed to continue only if the current input action matches the expected one. Of course, the match can be checked only *after* the component has received the data at its interface; after all, objects are *input enabled* in our semantics.

In case of incoming *calls,* i.e., in case of L-CallI of the construction, the situation is still a bit more complex: in this case, the code of the method is

[26] While the scripts will be kept in an instance variable of each object, conceptually they describe the future of the whole clique. The values of the scripts for each object will be kept in sync, i.e., we will maintain the *invariant* that all members of a clique agree upon their potential futures. Note further that while traces of a component are tree-structured, the futures are *linear*, since the trees branch into the past, since cliques only merge, but never split.

[27] To be more precise: we cannot program that the instantiation itself does the random choice, but it's part of the synchronization code for instantiation.

"wrapped up", i.e., in first approximation copied from the thread into the method body, which means, into the class. Of course, a method may be invoked more than just once in a legal trace and, when letting the constructed component run, all witnessed reactions must be possible and more stringently: for each occurrence of the call on the trace, the *corresponding* reaction must follow. This means the synchronisation code at the beginning of the method body must *dispatch* to the right piece of code when invoked.[28]

When an object accepts a method call, it compares this action to the yet-to-do script of the clique/object. If there is a matching one first in the script, it "works it off" as described above. The body of each method therefore is mainly a large conditional, and we generalize the guarded command syntax $\check{a} \rhd t$ to cover this branching:

$$\begin{array}{l} \mathit{case} \\ \quad \check{a}_1 \rhd t_{body_1} \\ \quad \ldots \\ \quad \check{a}_n \rhd t_{body_n} \\ \mathit{esac} \end{array}$$

Note that we do not require the branches to be disjoint. Informally, the intended behavior of the construct is as follows: assume the method l is invoked via the action $n\langle \mathit{call}\ o_2.l(\vec{v})\rangle?$. The current action is compared with the expected according to the *script* instance variable. If a is a possible next step, then *one* matching branch is chosen and the future is shortened by one. This must be done not just for the callee object, but for all members of the callee's clique. Since we assured at this point that the clique is script-consistent, the shortening is well-defined. Note further that the class stores in its fields the static variants $\check{a}\ \ldots$ for comparison.

$$n\langle \mathit{case} \ldots \check{a} \rhd t \ldots \mathit{esac}; t'\rangle \parallel o[\mathit{script} = \check{a} :: s, \ldots] \Longrightarrow n\langle t; t'\rangle \parallel o[\mathit{script} = s, \ldots]$$

To avoid unduly complicated concurrent behavior or even inconsistency of the data structures by interleaving of all interactions and data maniplation, all the described book-keeping is done under mutucal exclusion, at least per clique.[29] We use the syntax $(\!|t|\!)$ to indicate that the code t is executed without interference from other threads. Intuitively, the opening parenthesis $(\!|$ takes a lock (if available) which ensures undisturbed access to the whole clique.[30] The dual $|\!)$ releases the lock again. See below for details.

[28] The attentive reader will have noticed that Proposition 2 does not require that the constructed component shows no other reaction than the one given by the legal trace; it shows only, that indeed C_s realizes the trace. For completeness, as mentioned, we will need this exactitude.

[29] Two different cliques cannot be coordinated, of course, as they are unconnected by definition and enforcing mutex would require at least some bit of shared information.

[30] Locking the whole, distributed network of the clique objects looks harder than it is. Since we are after may-testing, only, we need not worry about deadlock, let alone

We start with the synchronization action, i.e., the code used in the cases for L-CallO and L-RetO. There, the code t^o_{sync} is "called" with the intended next action of the thread. The abstraction here is understood as meta-mathematical notation, only; we cannot hand-over class names to a method. Also *pick* in the definition is meant meta-mathematically: it calls a method (also) called *pick* for all instance variables from $\vec{x}$ whose type is a internal class. The exact definitions are given in the appendix.

Definition 6 (Synchronization, output). *Given a component $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ and furthermore $\vec{c}$ a number of class names, i.e., $\Theta, \Delta \vdash c_i : [\![\dots]\!]$, for all c_i. Additionally assume $\Theta; E_\Theta \vdash n \hookrightarrow o : \Theta$ with $\Theta \vdash n : thread$. Then*

$$t^o_{sync}(\vec{c}, \check{a}) \triangleq (\!| \; let\ \vec{x}{:}\vec{c} = new(\vec{c}) \\ in\ \ pick(\vec{x}); \\ \check{o}.propagate(\vec{x}); \\ \check{a} \rhd \; |\!) \ . \tag{16}$$

We will also use the following notation: Assume two object bindings Δ' and Θ' where $\Delta' = \vec{o}{:}\vec{c}$ and $\Theta' = \vec{o}'{:}\vec{c}'$. Then $t^o_{sync}(\Delta', \Theta', \check{a})$ stands for $t^o_{sync}(\vec{c}, \vec{c}', \check{a})$.

Note that in the definition, the newly created references are *first* propagated through the object graph, and only afterwards it is checked for script-consistency and whether the next action is the expected one using the guard $\check{a} \rhd \epsilon$. This is necessary since we base the comparison in the guard $\check{a}$ upon the respective instance variables (and not the local variables) and those are set by the propagate method. Note further that $\check{o}$ is the instance variable representing the object reference o, which corresponds to the "current object", since we assumed $n \hookrightarrow o$.

The following lemma expresses that the output synchronization does the expected job, i.e., it creates the required internal objects mentioned in Θ', initiates the objects to be lazily instantiated in the external step to follow from Δ', and takes care the full-connectivity is preserved.

Lemma 1 (Synchronization, output). *Given a fully-connected component $\Delta; E_\Delta \vdash_{\mathcal{S}} C \parallel n\langle t^o_{sync}(\Delta', \Theta', \check{a})\rangle : \Theta; E_\Theta$. Additionally $\Theta; E_\Theta \vdash n \hookrightarrow o : \Delta$ with $\Theta \vdash n : thread$. If the component is script-consistent and $o.script = \check{a} :: s'$ for some s', then*

$$\Delta; E_\Delta \vdash_{\mathcal{S}} C \parallel n\langle t^o_{sync}(\Delta', \Theta', \check{a}); t\rangle : \Theta; E_\Theta \Longrightarrow \\ \Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\Delta', \Theta').C' \parallel n\langle t\rangle : \Theta; E_\Theta \ .$$

Proof. With the help of Lemma **??** for locks and the propagation Lemma 33 for output. □

One case of synchronization requires special attention, namely when there is an outgoing communication where a new thread is made known to the environment. This can only happen for method calls. The code is similar to the one of

loosing liveness or even fairness. It suffices that any failed attempt to obtain the lock leads simply blocks or diverges.

Definition 6 except that it is now part of a thread-creation statement. Note also, that the expression *currentthread*, when being evalutated, refers to the newly created thread.

Besides the situation, when a component thread is spawned by another one already resident in the component, we also have to consider the case where the thread is present in the component from the start. In this case, the thread is not spawned by another one and there is no mechanism to "hand over" any values. In other words, the thread has no access to any values already known to the environment, and all values being exported by the call must not be acquainted to any other objects. We assume that this very particular situation is given only for one thread, the initial one.[31]

Definition 7 (Synchronization: Output call & thread creation). *Given a component* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ *and furthermore* $\vec{c}$ *a number of class names, i.e.,* $\Theta, \Delta \vdash c_i : [\![\ldots]\!]$*, for all* c_i*. Additionally assume* $\Theta; E_\Theta \vdash n \hookrightarrow o : \Theta$ *with* $\Theta \vdash n : thread$*. Then*

$$t^{ont}_{sync}(\vec{c}, \breve{a}) \triangleq (\!| \, new \; (\; let \; \vec{x}{:}\vec{c} = new(\vec{c}) \; in \; pick(\vec{x}); \; \breve{o}.propagate(currentthread, \vec{x}); \; \breve{a} \rhd |\!| \;) \; . \tag{17}$$

In case the new thread is the initial thread, the synchronization code needs not be created:

$$t^{int}_{sync}(\vec{c}, \breve{a}) \triangleq (\!| \; let \; \vec{x}{:}\vec{c} = new(\vec{c}) \; in \; pick(\vec{x}); \; \breve{o}.propagate(currentthread, \vec{x}); \; \breve{a} \rhd |\!| \; . \tag{18}$$

For inputs, we start with the simpler definition for returns. As can be seen in the construction for Proposition 2, especially Table 16, the code is used as abstraction over the bound variable(s) used to hold the incoming values. We define therefore t^{ir}_{sync} and further below t^{ic}_{sync} as abstraction which is again understood as meta-mathematical convention. The definition is slightly more general than necessary. The syntax of the calculus allows only a single return value, not a vector. We use the more general definition for uniformity.

Definition 8 (Synchronization, input return). *Given a fully-connected component* $\Delta; E_\Delta \vdash_{\mathcal{S}} C : \Theta; E_\Theta$*, and additionally* $\Theta; E_\Theta \vdash n \hookrightarrow o : \Delta$ *with*

[31] It would be straightforward to formulate the framework for more than one initial thread. However, all initially running threads will belong to separate cliques and will never be able to communicate or merge their cliques. A standard set-up therefore is to start with just one single thread. In *Java*, this is the body of the main method of the main class.

$\Theta \vdash n : \mathit{thread}$. *Then*

$$t^{ci}_{sync}(\vec{y}, \check{a}) \triangleq (\!| \; \mathit{pick}(\vec{y}); \\ \check{o}.\mathit{propagate}(\vec{x}); \\ \check{a} \rhd \; |\!) \; . \tag{19}$$

To formulate the reduction property concisely, we introduce further notation to denote the immediate next possible action as specified by the script instance variables, either of the script variable of the object itself or of the *scripts*-variable of the corresponding class.

Definition 9. *Assume a well-typed, script-consistent component* $\Delta; E_\Delta \vdash_{\mathcal{S}} C : \Theta; E_\Theta$. *Assume further a component object* $o[c, F]$ *with* $\Theta \vdash o : c$. *Then we write* $\Delta; E_\Delta \vdash_{\mathcal{S}} C : \Theta; E_\Theta :: o \rhd \check{a}$ *if one of the following two conditions hold:*

1. $o[c, F] = o[c, \mathit{script} = \check{a} :: s', \dots]$ *for some* s'.
2. $o[c, F] = o[c, \mathit{script} = \bot, \dots]$ *and* $\check{a} :: s' \in c.\mathit{scripts}$ *for some* s'.

For a tuple $\vec{v}$ *of values, we mean by* $\vec{v} \rhd \check{a}$ *that* $v \rhd \check{a}$ *for all references* $v \in \vec{v}$ *where* $\Theta' \vdash v : c$ *for some internal class name* c *with* $\Theta' \vdash c : [\![\dots]\!]$; *for references to external classes, nothing is required.*

Lemma 2 (Synchronization, input return). *Assume a well-typed, fully-connected component* $\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C' \parallel n\langle \mathit{let}\, x{:}T = t^{ir}_{sync}(\vec{v}, \check{a}); t' \rangle : \acute{\Theta}, E_\Theta$, *where* $\Theta; E_\Theta \vdash n \hookrightarrow o : \Delta$ *and* $\Theta \vdash o : c$ *and let* $\acute{E}_\Theta = E_\Theta + o \hookrightarrow \vec{v}$. *Assume further that the component is script-consistent and* $o, \vec{v} \rhd \check{a}$. *Then*

$$\acute{\Delta}; \acute{E}_\Delta \vdash C' \parallel n\langle \mathit{let}\, x{:}T = t^{ir}_{sync}(\vec{v}, \check{a}); t' \rangle : \acute{\Theta}, E'_\Theta \Longrightarrow \\ \acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C' \parallel n\langle t' \rangle : \acute{\Theta}; \acute{E}_\Theta$$

Proof. With the help of Lemma **??** for locks and the propagation Lemma 32 for input. □

In principle, the task for synchronization for incoming calls resembles the previous cases, especially the case for input returns: store and propagate the arguments in instance variable, choose a future for lazily instantiated objects, check for script consistency and whether the next action is the expected one, while avoiding interfernce by using the lock. What is different now is that the synchronization code, together with the thread realizing the intended behavior, is now to be found in the method body, i.e., in the respective class.

As a consequence, the synchronization code not only has to take care of mutual exclusion and information propagation, it has also to *dispatch* to the right piece of code in the method body. In the backward construction for Proposition 2, we abstractly described the required code transformation by the $\oplus$-operator (cf. especially Equation 12).

First we give the synchronization code t^{ic}_{sync} to field incoming calls. It resembles the code for treating incoming returns or the code for t^{o}_{sync}. There are two salient differences to $t^{ir}_{sync}(x, \check{a})$ from Definition 8, however. Being programmed

as part of a class, it contains hard-coded the dispatch code over all known alternatives. Additionally, as static class code, the body cannot refer to the (static representative) of the object's identity which is not yet available. It uses directly the local variable *self* instead. Note further that the callee object can itself be lazily instantiated during the call in which case also its future is chosen. Therefore, *self* is mentioned in the *pick*-code.

Definition 10 (Synchronization, input call).

$$
\begin{aligned}
t^{ci}_{sync}(self)(\vec{x}) \triangleq (\!| \; & pick(self, \vec{x}); \\
& self.propagate(\vec{x}); \\
& case \\
& \quad \check{a}_1 \rhd |\!|; t_{body_1} \\
& \quad \dots \\
& \quad \check{a}_n \rhd |\!|; t_{body_n} \\
& esac
\end{aligned}
\tag{20}
$$

More rigorously, we refer to the synchronization code t^{ic}_{sync} not as the full method body, but as the part between the $(\!|$- and $|\!)$-brackets, only. Therefore, stictly speaking, the code for input synchronization is defined as abstraction (on the meta-mathematical level) $t^{ic}_{sync}(\vec{x}, \check{a}_1, \dots, \check{a}_n, t_{body_n}, \dots, t_{body_n})$, i.e., abstracted not only over *self* and the parameters, but also over the alternatives.

Lemma 3 (Synchronization, input call). *Assume a well-typed, fully-connected component* $\acute{\Delta}; \acute{E}_\Delta \vdash_{\mathcal{S}} C : \acute{\Theta}, E_\Theta$ *where* $n \hookrightarrow o$*, and let* $\acute{E}_\Theta = E_\Theta + o \hookrightarrow \vec{v}$*. Assume further that the component is script-consistent with* $\acute{\Delta}; \acute{E}_\Delta \vdash C : \acute{\Theta}; \acute{E}_\Theta :: o, \vec{v} \rhd \check{a}_i$*. Then*

$$
\begin{aligned}
& \Delta; E_\Delta \vdash n\langle t^{ir}_{sync}(o, \vec{v}, \dots, \check{a}_i, \dots, t_{body_i}, \dots); t\rangle : \acute{\Theta}; \acute{E}_\Theta \stackrel{a}{\Longrightarrow} \\
& \Delta; E_\Delta \vdash_{\mathcal{S}} n\langle t_{body_i}; t\rangle : \Theta; E_\Theta \ .
\end{aligned}
$$

Proof. Using the propagation Lemma 32. □

Now the $\oplus$-transformation for classes, used in the cases for L-CALLI in the backward construction of C_s, is readily defined: it simply adds another alternative to the method body. A further code transformation for classes, also denoted by $\oplus$ is done at instantiation time, where the whole future of the object, collected during the backward construction, is added to the set *scripts* of possible futures of the class. Thus, the operation is the backward dual of the *pick*-code. Note that the instance variable *script* contains only *static* values in the construction, i.e., no reference values. Therefore we can just use the value of *script* as is in the construction of *scripts*.

Definition 11 ($\oplus$ on classes). *Given a class $c[\![O]\!]$ of the form*

$$\begin{array}{l} c[\![\ldots\ldots l = \varsigma(s{:}c)\lambda(\vec{x}{:}\vec{T})\,(\!|\ pick(\vec{x}) \\ \qquad\qquad\qquad\qquad\quad s.propagate(\vec{x}); \\ \qquad\qquad\qquad\qquad\quad case \\ \qquad\qquad\qquad\qquad\qquad \check{a}_1 \rhd |\!|); t_{body_1} \\ \qquad\qquad\qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad\qquad \check{a}_n \rhd |\!|); t_{body_n} \\ \qquad\qquad\qquad\qquad\quad esac \qquad\qquad\qquad |\!|]\!] \end{array}$$

with $n \geq 0$. Then $c[\![O]\!] \oplus (\check{a}, t_{body})$ is defined as

$$\begin{array}{l} c[\![\ldots l = \varsigma(s{:}c)\lambda(\vec{x}{:}\vec{T})\,(\!|\ pick(\vec{x}) \\ \qquad\qquad\qquad\qquad\quad s.propagate(\vec{x}); \\ \qquad\qquad\qquad\qquad\quad case \\ \qquad\qquad\qquad\qquad\qquad \check{a}_1 \rhd |\!|); t_{body_1} \\ \qquad\qquad\qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad\qquad \check{a}_n \rhd |\!|); t_{body_n} \\ \qquad\qquad\qquad\qquad\qquad \check{a} \quad \rhd |\!|); t_{body} \\ \qquad\qquad\qquad\qquad\quad esac \qquad\qquad\qquad |\!|]\!] \ . \end{array}$$

Overloading the symbol $\oplus$, $c[\![\ldots scripts = S \ldots]\!] \oplus script$ is defined by replacing S by $S \cup script$.

The final piece in the construction is the handling of the *script*-variable of objects. As mentioned, each object stores in *script* its future, which is kept consistent for all members of a clique at each moment. Thus in the backward construction of C_s, the *script*-variable of all members of a clique are extended at the head with this action (resp. the static variant of it).

Definition 12 ($\oplus$ on objects). *Given an object $o[script = s, \ldots]$, a static variant $\check{a}$ of an action, then*

$$o[script =, \ldots] \oplus \check{a} \triangleq o[script = \check{a} :: script, \ldots]$$

In the construction of Proposition 2, the $\oplus\check{a}$-operation is applied to *all* members of a clique. Thus the construction preserves script-consistency.

Theorem 1 (Definability). *Assume $\Delta; E_\Delta \vdash s : trace\,\Theta; E_\Theta$. Then $\Delta; E_\Delta \vdash C_s : \Theta; E_\Theta \stackrel{s'}{\Longrightarrow}$ if and only if $\Delta; E_\Delta \vdash s' \sqsubseteq s : \Theta; E_\Theta$.*

Theorem 2 (Completeness). *If $\Delta \models C_1 \sqsubseteq_{may} C_2 : \Theta$, then $[\![C_1]\!]_{trace} \subseteq [\![C_2]\!]_{trace}$.*

7 Conclusion

In this report we presented, as an extension of the work of [JR02], 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 [JR02] to a framework with classes requires quite some extension in the operational semantics to characterize the possible behavior of a component. In particular it is necessary to keep track of the potential *connectivity* of objects of the environment to exclude impossible communication labels.

It is therefore instructive to review the differences in this conclusion and explain them from a higher perspective, especially trying to understand how the result of [JR02] can be understood as a special case of the framework explored here.

The fundamental dichotomy underlying the observational definition of equivalence is the one between the inside and the outside: program or component vs. environment or observer. This leads to the crucial difference between object-based languages, instantiating from objects, and class-based language, instantiating from classes: In the class-based setting, instantiation may *cross the demarcation line between component and environment,* while in the object-based setting, this is not possible: the program only instantiates program objects, and the environment only objects belonging to the environment. All other complications, 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 [JR02], 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 by invariantly (and trivially) taking $E_\Delta = \Delta \times (\Delta + \Theta)$, while in our setting, we take into account that E_Δ may be more specific.

Related work The work closest to ours clearly is the one of Jeffrey and Rathke [JR02] about full abstraction in the original, i.e., object-based calculus. In effect, the current paper can be seen as a direct extension of their work; not only do we borrow much from the notation found there, also the general set-up of the proof follows their path.

Nonetheless, the seemingly innocent change from an object-based to a class-based setting entailed an all but trivial change in the denotational, fully abstract semantics, notably by taking *connectivity* amongst objects into account, unnecessary in the object-based setting. 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 [JR02] can be understood as a special case of the framework explored here.

The fundamental dichotomy underlying the observational definition of equivalence is the one between the inside and the outside: program or component vs. environment or observer. This leads to the crucial difference between object-based languages, instantiating from objects, and class-based language, instantiating from classes: In the class-based setting, instantiation may *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 complications, expounded here at some great length, follow from this difference.

The most visible complication is that it turns out necessary to represent the dynamic object structure into the semantics, or rather an approximation of the connectivity of the environment objects.

[Vis98] investigates the full-abstraction problem in an object calculus with subtyping. The setting is a bit different from the more standard one as used here as he does not compare an contextual semantics with an denotational one, but a semantics by translation by a direct one. The paper considers neither concurrency nor aliasing.

[FMS96] present a full-abstraction result for the π-calculus, the standard process algebra for name passing and dynamically changing process structures. The extensional semantics is given as a domain-theoretic, categorical model, and using bisimulation equivalence as starting point, not may testing resp. traces as here. [Yos96] gives equational full abstraction for standard translation of the polyadic π-calculus into the monadic one. Without additional information, the translation is not fully-abstract, and [Yos96] introduces graph-types as an extension to the π-calculus sorting to achieve full abstraction. The graph types abstracts the *dynamic* behavior of processes. In capturing the dynamic behavior of interaction, Yoshida's graph types are rather different from the graph abstracting the connectivity of objects presented here.

Future work We imagine to extend the language and result in a number of way. 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 objects, 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 instantiation, 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.

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.

Acknowledgements We would like to thank Karsten Stahl for "active listening" even to the more byzantine details and dead ends of all this. Thanks likewise

to Willem-Paul de Roever for careful reading and spotting many sloppy points and to Uwe Nestmann for pointing out and discussing related work. We are also indepted to Ben Lukoschus for helping with some of the more arcane TEX-stunts. Thanks likewise to Andreas Grüner for careful reading and improving a number of half-baken previous versions of the document. Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2).

References

ÁBdBS03. Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, August 2003.

AC96. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

FMS96. Marcelo Fiore, Eugenio Moggi, and Davide Sangiorgi. A fully-abstract model for the π-calculus (extended abstract). In *Proceedings of LICS '96*, pages 43–54. IEEE, Computer Society Press, July 1996.

GH98. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

GHL97. A. D. Gordon, P. D. Hankin, and S. B. Lassen. Compilation and equivalence of imperative objects. In S. Ramesh and G. Sivakumar, editors, *Proceedings of FSTTCS '97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, December 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.

Hen88. Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

JR02. Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.

MPW92. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.

MS92. Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Proceedings of ICALP '92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer-Verlag, 1992.

PS93. A. M. Pitts and D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, September 1993.

SW01. Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

Vis98. Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.

Yos96. Nobuko Yoshida. Graph types for monadic mobile processes. In V. Chandru and V. Vinay, editors, *Proceedings of FSTTCS '96*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996. Full version as Technical Report ECS-LFCS-96-350, University of Edinburgh.

ZP97. Xiaogang Zhang and John Potter. Class-based models in the pi-calculus. In Christine Mingins, Roger Duke, and Bertrand Meyer, editors, *Proceeding of The 25th International Conference in Technology of Object-Oriented Languages and Systems (TOOLS Pacific '97, Melbourne, Australia)*, pages 219–231, November 1997.

A Proofs

The appendix collects proofs omitted from the main part of the paper and additional lemmas.

A.1 Operational semantics

Lemma 4 (Disjointness of object and class references). *Assume* $\Delta_1; E_{\Delta_1} \vdash C_1 : \Theta_1; E_{\Theta_1}$ *For all reachable configurations,* $dom(\Delta) \cap dom(\Theta) = \emptyset$ *for all object and class references.*

Proof. By induction on the length of reduction. Internal steps and the rules for structural congruence leave the contexts untouched. The external steps dealing with scoping from Table 10 add a fresh *object* names only to *either* Δ or to Θ, and the freshness assumption assures that the new name does not occur on both contexts. Class names are never exchanged bound. Only the rules for extending the scope of thread identifiers BIN_{thread} and BOUT_{thread} lead to a non-empty intersection of the domains of Δ and Θ. □

Lemma 5 (Static nature of class names). *If* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$*, then for all class names* c*,* $c \in dom(\Delta)$ *iff.* $c \in dom(\acute{\Delta})$ *and likewise* $c \in dom(\Theta)$ *iff.* $c \in dom(\acute{\Theta})$*.*

Proof. Straightforward. Class names cannot be sent around, hence they never occur in a communication label. □

The following lemma states that well-typedness of a component is preserved under reduction. This property is also known as *subject reduction.*

Lemma 6 (Subject reduction). *Assume* $\Delta, E_\Delta \vdash C : \Theta, E_\Theta$*.*

1. *(a) If* $C \rightsquigarrow C'$*, then* $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$*.*
   (b) If $C \xrightarrow{\tau} C'$*, then* $\Delta, E_\Delta \vdash C' : \Theta; E_\Theta$*.*
   (c) If $C \equiv C'$*, then* $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$*.*
2. *If* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \Delta'; E'_\Delta \vdash C' : \Theta'; E'_\Theta$*, then* $\Delta'; E'_\Delta \vdash C' : \Theta'; E'_\Theta$*.*

status: Not done yet. We assume it's ok.

Proof. All parts by induction on the length of derivation for the reduction step. □

Lemma 7.

We call a well-typed component $\Delta \vdash C : \Theta$ (and analogously for components $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$) is *instance closed,* if for all identifiers o with $; \Theta \vdash o : c$, also $; \Theta \vdash c : T$. In other words, each object identifier typable in Θ and thus occurring free in the component C, is an instance of a class also typable in Θ. Note that the type systes assures that T is a class type, i.e., $T = [\![T']\!]$. For example, $\vdash o[O'] \parallel c[\![O]\!] : o{:}c, c{:}[\![T]\!]$ is instance closed, but the component containing the object o in isolation is not. Instance closedness is preserved under reduction.

Lemma 8 (Preservation of instance closedness). *Assume* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{a}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$*. If* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ *is instance closed, then so is* $\acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$*.*

Proof. Straightforward, using subject reduction: Internal steps do not change the contexts nor do they change the externally visible classes or objects. The same holds for the structural rules. The external rules maintain instance closedness by distinguishing in the exchange of bound (object) references according to the class: Rule BIN_{new} for an incoming lazy instantiation is enabled only if an instance of a component class is sent. Lazy instantiation in outbound direction is described by the internal rule NEWO_{lazy} (in combination with BOUT_{new}). Also here, the premises of the rules assure that o as instance of an environment class is not instantiated internally. The rule for non-lazy scope extrusion BIN adds the new objects' identity to the assumption context, only. For BOUT, scope extrusion is achieved in combination with the non-lazy interal instantiation rule NEWO_i from Table 5 which assure that both the object after the step and the class are part of the component. The rule COMM and the ones dealing with scope extrusion for a thread names finally leave the name contexts unchanged, at least wrt. object and class names. □

In the whole development we will always assume that well-typed components are instance closed. Under this assumption we will also used the following abbreviation. For $\Delta \vdash o : c$ and $\Delta \vdash c : T$, we write shorter $\Delta \vdash o :: T$, when the name of the class c does not play a role.

The assumption and commitments context play, not surprisingly, a dual role in the semantics. For instance, in case of incoming communication (cf. for instance rule CALLI), the assumption context is checked as premise, the commitment context is updated. In connection with the exchange of bound names, however, also the assumption context is updated. However, this does not lead to new information about names already known. Again in the situation for incoming communication: 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, initially unknown to the component, the environment cannot contact objects it could not contact otherwise. In other words, the connectivity assumption context $\acute{E}'_\Delta$ after an incoming communication is a *conservative* extension of E_Δ wrt. the old objects.

The property that the addition of connectivity of the newly received indentities added to Δ may not lead to new derivable equations for the objects previously known is formulated in the following lemma using a projection operator which simply resticts the graph of $\acute{E}_\Delta$ to the domain $\Delta \times (\Delta + \Theta)$. Thus, $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{\gamma?}{\longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash C : \acute{\Theta}; \acute{E}_\Theta$ stands for the implication:

$$\text{If } \Delta'; E'_\Delta \vdash o_1 \leftrightharpoons\!\hookrightarrow o_2 : \Theta, \text{ then } \Delta; E_\Delta \vdash o_1 \leftrightharpoons\!\hookrightarrow o_2 : \Theta \ ,$$

for all o_1 from Δ and o_2 from Δ, Θ. We can thus state.

Lemma 9 (Conservativity of connectivity information). *Assume a input step* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash C : \acute{\Theta}; \acute{E}_\Theta$. *Then*

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

The situation for outgoing information is dual.

Proof. Straightforward. □

A.2 Soundness

The merge of two components respectively traces is defined as follows:

Definition 13 (Merge). *For a pair of components respectively for a pair of threads,* $\between$ *is the symmetric, partial operator up-to* $\equiv$ *defined by Table 17, where in the last case for* $t_1 \between t_2$, *the expression* e *is block/return free and* $y \notin fv(t_2)$.

The definition is essentially the same as in [JR02]. The one minor difference concerns $t_1 \between t_2$ in the case of merging a return statement with a blocked partner, as the stack structure differes slightly in our situation as we need to remember the caller of a method.

Note that the $\between$-operation is partial, i.e., it may fail if the two participating components and in particular two equally named threads cannot be combined into a common stack. The rules for $C_1 \between C_2$ in the first part of the table are mostly straightforward. For the empty component, there is nothing to merge. Objects and classes do not participate in the merging, and likewise the scoping operator is ignored. Two equally named threads are merged and the rest of the components are merged recursively in the last equation.

For two threads, the $\between$-operator is defined by case analysis on the outermost let-construct (if there thread is not completely stopped). This corresponds to the top-most part of the common stack for both threads. If one if the threads is stopped, the combination of both is stopped, as well, since the *stop*-expression unconditionally terminates a thread irrespective of trailing statements (cf. rule Stop of Table 5). If a part of a thread is blocked in its topmost construct, it indicates that it waits for a return of the activity from its partner (cf. rule CallOut). In case the partner is a value, it means, the return will never happend and the blocked part can be discarded. If otherwise the partner is about to perform the return, the values is handed over, the two topmost let-bindings are thereby popped off, and the merge-operator recurs through the rest of the two threads. The last equation, finally, deals with the situation that the partner is not (yet) a value or a return statement and rearranges the let-binding appropriately.

As far as objects and classes are concerned, merging of two components behaves like their parallel composition:

Lemma 10 ($\parallel$ and $\between$). *If* $\Delta; E_\Delta \vdash C_1 \parallel C_2 : \Theta; E_\Theta$, *then* $C_1 \parallel C_2 \equiv C_1 \between C_2$.

Proof. By induction on the definition of $\between$, i.e., the left-hand sides of the equations as given in Table 17.

Case: $\mathbf{0} \between C$
Directly by the corresponding rule for $\|$ from Table 6.

Case: $((\nu n{:}T).C_1) \between C_2$,
where $n \notin fn(C_2)$. The merge is then given by $(\nu n{:}T).(C_1 \between C_2)$, which by induction is equivalent to $(\nu n{:}T).(C_1 \parallel C_2)$ which furthermore yields $((\nu n{:}T).C_1) \parallel C_2)$, since $n \notin fn(C_2)$.

Case: $(o[c] \parallel C_1) \between C_2$, $(c[O] \parallel C_1) \between C_2$ and $(n\langle t\rangle \parallel C_1) \between C_2$
By straightforward induction, using associativity of the $\|$-operator.

Case: $(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2)$
In this case, the component $(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2)$ is not well-typed (cf. especially rules T-Par and T-NThread from Table 3. □

For named threads, occurring on both sides of the $\between$-operator, the respective stack framed are "zipped" into a common stack (if possible). As a consequence: If $C = C_1 \between C_2$ is defined, then an extenal step of a thread n is enabled for C exactly if the step is enabled for C_1 or for C_2. For the special case of barbing, this is expressed in the following lemma.

Lemma 11 (Merging and barbing). *Assume $C \equiv C_1 \between C_2$. Then $C \downarrow_{c_b}$ iff. $C_1 \downarrow_{c_b}$ or $C_2 \downarrow_{c_b}$.*

Proof. By induction on the definition of $C_1 \between C_2$, where we omit symmetric cases. Before we start, recall the definition of barbing, stipulating that $C \downarrow_{c_b}$ if C is structurally congruent to $\nu(\vec{n}{:}\vec{T}, b{:}\mathsf{barb}).\ C' \parallel n'\langle let\ x{:}none\ in\ b.succ()\ in\ t\rangle$.

Case: $C_1 \between \mathbf{0}$
Since $\mathbf{0}$ is inert, directly $C_1 \downarrow_{c_b}$.

Case: $C = (\nu(n{:}T).C_1) \between C_2 \equiv \nu(n{:}T).(C_1 \between C_2)$,
where $n \notin dom(C_2)$. Assume $C \downarrow_{c_b}$, i.e.,

$$\nu(n{:}T).(C_1 \between C_2) \equiv \nu(\vec{n}{:}\vec{T}, b{:}\mathsf{barb}).\ C' \parallel n'\langle let\ x{:}none\ in\ b.succ()\ in\ t\rangle$$

it is of the form For the "only-if" direction assume

Case: $C = (o[O] \parallel C_1) \between C_2 \equiv o[O] \parallel (C_1 \between C_2)$
Assume $C \downarrow_{c_b}$, i.e.,

$$o[O] \parallel (C_1 \between C_2) \equiv \nu(\vec{n}{:}\vec{T}, b{:}\mathsf{barb}).\ C' \parallel n'\langle let\ x{:}none\ in\ b.succ()\ in\ t\rangle\ .$$

This implies that $(C_1 \between C_2) \downarrow_{c_b}$. Hence by induction, $C_1 \downarrow_{c_b}$ or $C_2 \downarrow_{c_b}$, and thus $(C_1 \parallel o[c]) \downarrow_{c_b}$ or $C_2 \downarrow_{c_b}$, as required.

For the reverse direction, assume $(o[O] \parallel C_1) \downarrow_{c_b}$, which implies $C_1 \downarrow_{c_b}$, and thus by induction $(C_1 \between C_2) \downarrow_{c_b}$, from which the case follows. The argument for the second alternative, starting from $C_1 \downarrow_{c_b}$, is similar

Case: $(c[\![O]\!] \parallel C_1) \between C_2 \equiv c[\![O]\!] \parallel (C_1 \between C_2)$
Analogous.

Case: $(n\langle t\rangle \parallel C_1) \between C_2 \equiv n\langle t\rangle \parallel (C_1 \between C_2)$,
where $n \notin dom(C_2)$. Analogous.
Case: $(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2) \equiv n\langle t_1 \between t_2\rangle \parallel (C_1 \between C_2)$,
where $n \notin dom(C_2)$. First assume, $(C_1 \between C_2) \downarrow_{c_b}$. Then the case follows by induction, as in the previous cases. The only interesting case is when $C' = C_1 \between C_2$ and $n = n'$, i.e.,

$$n\langle t_1 \between t_2\rangle = n\langle let\ x{:}none\ in\ b.succ()\ in\ t\rangle\ .$$

In this case, the last clause of Table 17 applies, so that $n\langle t_1\rangle = n\langle let\ y{:}T = o_1\ blocks\ for\ o_2\ in\ t_1'\rangle$ and $n\langle t_2\rangle = n\langle let\ y{:}none = b.succ()\ in\ t_2'\rangle$, which means $(n\langle t_1\rangle \parallel C_1) \downarrow_{c_b}$. The reverse direction is analogous. □

Lemma 12 ($\between$ and $\rightsquigarrow$-step). *If $C_1 \between C_2 \equiv C$ and $C_1 \rightsquigarrow \acute{C}_1$, then $C \rightsquigarrow \acute{C}$ with $\acute{C}_1 \between C_2 \equiv \acute{C}$. Pictorially:*

$$\begin{array}{ccc} C_1 \between C_2 & \equiv & C \\ \wr\downarrow & & \wr\downarrow \\ \acute{C}_1 \between C_2 & \cdots & \acute{C}\ . \end{array}$$

Proof. By induction on the length of derivation for $C_1 \rightsquigarrow \acute{C}_1$, using the operational axioms from Table 5 and the rules from Table 7, with the help of the rules for structural congruence from Table 6. □

Lemma 13 ($\between$ and τ-step). *If $C_1 \between C_2 \equiv C$ and $C_1 \xrightarrow{\tau} \acute{C}_1$, then $C \xrightarrow{\tau} \acute{C}$ with $\acute{C}_1 \between C_2 \equiv \acute{C}$. Pictorially:*

$$\begin{array}{ccc} C_1 \between C_2 & \equiv & C \\ \downarrow\tau & & \vdots\tau \\ \acute{C}_1 \between C_2 & \cdots & \acute{C}\ . \end{array}$$

Proof. Analogous to the proof of Lemma 12. □

Lemma 14. *Assume $\Delta, \Phi; E_\Delta \vdash C_1 : \Theta, \Psi; E_\Theta$ and $\Theta, \Phi; E_\Theta \vdash C_2 : \Delta, \Psi; E_\Delta$ where $C_1 \between C_2 \equiv C$. If $\Delta, \Phi; E_\Delta \vdash C_1 : \Theta, \Psi; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}, \Phi; \acute{E}_\Delta \vdash \acute{C}_1 : \acute{\Theta}, \acute{\Psi}; \acute{E}_\Theta$ and $\Theta, \Phi; E_\Theta \vdash C_2 : \Delta, \Psi; E_\Delta \xrightarrow{\gamma!} \acute{\Theta}, \Phi; \acute{E}_\Theta \vdash \acute{C}_2 : \acute{\Delta}, \acute{\Psi}; \acute{E}_\Delta$. Then $C \equiv \nu(\acute{\Sigma} \setminus \Sigma).\ \acute{C}_1 \between \acute{C}_2$, where $\Sigma = \Delta, \Theta; \Psi$ and $\acute{\Sigma} = \acute{\Delta}, \acute{\Theta}, \acute{\Psi}$. Pictorially*

$$\begin{array}{lllll} \Delta, \Phi; E_\Delta \vdash C_1 & : \Theta, \Psi; E_\Theta & \xrightarrow{\gamma?} & \acute{\Delta}, \Phi; \acute{E}_\Delta \vdash \acute{C}_1 & : \acute{\Theta}, \acute{\Psi}; \acute{E}_\Theta \\ \Theta, \Phi; E_\Theta \vdash & C_2 : \Delta, \Psi; E_\Delta & \xrightarrow{\gamma!} & \acute{\Theta}, \Phi; \acute{E}_\Theta \vdash & \acute{C}_2 : \acute{\Delta}, \acute{\Psi}; \acute{E}_\Delta \\ \downarrow & \downarrow & & \downarrow & \downarrow \\ C_1 \between C_2 & \equiv & & \nu(\acute{\Sigma} \setminus \Sigma).\ \acute{C}_1 \between \acute{C}_2) & \end{array}$$

Proof. By case analysis on the form of the communication label γ.

Case: $\gamma = \nu(\Delta', \Theta', n\colon thread).n\langle call\ o_2.l(\vec{v})\rangle$
i.e., $n \notin dom(\Psi)$. By the operational rules of Tables **??** and **??** and the typing rules, the two components are of the following form: C_1 does not contain the thread n and

$$C_2 \equiv \nu(n\colon thread, \Delta', \Theta', \vec{n}{:}\vec{T})(C_2' \parallel n\langle let\ x{:}T = [o_1]o_2.l(\vec{v})\ in\ t\rangle)\ .$$

By convention, Δ' contains objects from C_2 and Θ' those lazily instantiated to be contained in C_1 from this communication step on. After $\gamma?$, respectively $\gamma!$, we get

$$\acute{C}_1 = C_1 \parallel n\langle let\ y{:}T = o_2.l(\vec{v})\ in\ o_2\ return\ to\ o_1\ y\rangle$$

and

$$\acute{C}_2 = \nu(\vec{n}{:}\vec{T}).\ (C_2' \parallel n\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle)\ .$$

and for the contexts after the step $\acute{\Delta} = \Delta, \Delta'$, $\acute{\Theta} = \Theta, \Theta'$, and $\acute{\Psi} = \Psi, n\colon thread$. Since the thread n is not contained in C_1 and likewise the names from Δ', Θ', and $\vec{n}$ are new for C_1, we get by definition of the merge operator:

$$C_1 \between C_2 \equiv \nu(\Delta', \Theta', n\colon thread, \vec{n}{:}\vec{T}).\ (C_1 \between C_2' \parallel n\langle let\ x{:}T = [o_1]o_2.l(\vec{v}\ in\ t)\rangle)\ .$$

For the components $\acute{C}_1$ and $\acute{C}_2$ after the common step, the last two clauses in the definition of merging yield

$$\acute{C}_1 \between \acute{C}_2 \equiv \nu(\vec{n}{:}\vec{T}).(C_1 \between C_2'' \parallel n\langle let\ y{:}T = [o_1]o_2.l(\vec{v})\ in\ t[y/x]\rangle)\ ,$$

which means

$$C \equiv \nu(\acute{\Sigma} \setminus \Sigma)\ (\acute{C}_1 \between \acute{C}_2)\ ,$$

as required.
Case: $\gamma = \nu(\Delta', \Theta').n\langle call\ o_2.l(\vec{v})\rangle$,
with $n \notin dom(\Delta', \Theta')$. Similar to the previous case.
Case: $\gamma = \nu(\Delta', \Theta').n\langle return(v)\rangle$
Note that unlike the cases for method calls, the thread name cannot be transmitted boundedly. Furthermore, if we only transmit a single (non-compound) value v and not a vector as for method calls, then Δ' or Θ' are empty. For uniformity, we treat them both in one case. By the operational rules of Tables **??** and **??**, the components must be of the following forms:

$$\begin{aligned} C_1 &\equiv \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle t_1\rangle \\ &= \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t_1'\rangle \end{aligned}$$

and

$$\begin{aligned} C_2 &\equiv \nu(\Delta', \Theta', \vec{n}_2{:}\vec{T}_2).\ C_2' \parallel n\langle t_2\rangle \\ &\equiv \nu(\Delta', \Theta', \vec{n}_2{:}\vec{T}_2).\ C_2' \parallel n\langle let\ x{:}T = o_2\ return\ to\ o_1\ v; t_2'\rangle \end{aligned}$$

After the communication, the components look as follows (cf. rules RetI and RetO):

$$\acute{C}_1 \equiv \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle t_1'[v/x]\rangle \quad \text{and} \quad \acute{C}_2 \equiv \nu(\vec{n}_2{:}\vec{T}_2).\ C_2' \parallel n\langle t_2'\rangle\ .$$

For the change of contexts, we get $\acute{\Delta} = \Delta, \Delta'$, $\acute{\Theta} = \Theta, \Theta'$, and Ψ containing the thread names remains unchanged, i.e., $\acute{\Psi} = \Psi$.

Since the names from $\vec{n}_2$, Δ', and from Θ' are new for C_1 and $\vec{n}_1$ new for C_2, the definition of $\between$ for components and for threads (and using symmetry) gives (we assume that $\between$ has a stronger binding power than $\parallel$):

$$\begin{aligned}
C_1 \between C_2 &\equiv \nu(\Delta', \Theta', \vec{n}_1{:}\vec{T}_1, \vec{n}_2{:}\vec{T}_2).\ (C_1' \between C_1' \parallel n\langle t_1 \between t_2\rangle)\\
&\equiv \nu(\Delta', \Theta', \vec{n}_1{:}\vec{T}_1, \vec{n}_2{:}\vec{T}_2).\ (C_1' \between C_1' \parallel n\langle t_1'[v/x] \between t_2'\rangle)\\
&\equiv \nu(\Delta', \Theta').\ \nu(\vec{n}_1{:}\vec{T}_1, \vec{n}_2{:}\vec{T}_2).\ (C_1' \between C_1' \parallel n\langle t_1'[v/x] \between t_2'\rangle)\\
&\equiv \nu(\acute{\Sigma} \setminus \Sigma).\ (\nu(\vec{n}_1{:}\vec{T}_1).\ (C_1' \parallel n\langle t_1'\rangle[v/x]) \between \nu(\vec{n}_1{:}\vec{T}_1).\ (C_2' \parallel n\langle t_2'\rangle))\ .\\
&= \nu(\acute{\Sigma} \setminus \Sigma).\ (\acute{C}_1 \between \acute{C}_2)\ ,
\end{aligned}$$

which concludes the case. □

Lemma 15 (Trace composition). *Assume $\Delta, \Phi \vdash C_1 : \Theta, \Psi$ and $\Theta, \Phi \vdash C_2 : \Delta, \Psi$ with $C_1 \between C_2 \equiv C$. If $\Delta, \Phi \vdash C_1 : \Theta, \Psi \stackrel{s}{\Longrightarrow} \Delta', \Phi \vdash C_1' : \Theta', \Psi'$ and $\Theta, \Phi \vdash C_2 : \Delta, \Psi \stackrel{\bar{s}}{\Longrightarrow} \Theta', \Phi \vdash C_2' : \Delta', \Psi'$, then $C \Longrightarrow C'$ where $C' \equiv \nu(\Delta', \Theta', \Psi' \setminus \Delta, \Theta, \Psi).\ C_1' \between C_2'$.*

Proof (of trace composition (Lemma 15)). By induction on the length of reduction (cf. Table 11), using subject reduction and the Lemmas 12 and 13 dealing with $\rightsquigarrow$-steps resp. τ-steps of one of the partners, and Lemma 14, dealing with communication between the partners, resolved in a common τ-step. □

Lemma 16 (Prefix).

Proof. Obvious □

A.2.1 Trace decomposition

Lemma 17. *If $\Delta, \Phi; E_\Delta \vdash C_1 : \Theta, \Psi; E_\Theta$ and $\Theta, \Phi; E_\Theta \vdash C_2 : \Delta, \Psi; E_\Delta$ where $C_1 \between C_2 \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t\rangle)$, then*

$$\Delta, \Phi; E_\Delta \vdash C_1 : \Theta, \Psi; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}, \Phi; \acute{E}_\Delta \vdash \acute{C}_1 : \acute{\Theta}, \acute{\Psi}; \acute{E}_\Theta$$

with $\acute{C}_1 = \nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle \mathit{let}\ x{:}T = e\ \mathit{in}\ t_1\rangle$ and

$$\Theta, \Phi; E_\Theta \vdash C_2 : \Delta, \Psi; E_\Delta \stackrel{\bar{s}}{\Longrightarrow} \acute{\Theta}, \Phi; \acute{E}_\Theta \vdash \acute{C}_2 : \acute{\Delta}, \acute{\Psi}; \acute{E}_\Delta$$

and where $\nu(\acute{\Sigma} \setminus \Sigma).\nu(\vec{n}_1{:}\vec{T}_1)(C_1' \parallel n\langle t_1\rangle) \between \acute{C}_2 \equiv \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle t\rangle)$. *Pictorially*

$$\begin{array}{ccc} C_1 \between C_2 & \equiv\equiv\equiv & \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle let\ x{:}T = e\ in\ t\rangle) \\ \Big\downarrow{\scriptstyle \bar{s}\,s} & & \\ (\nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle let\ x{:}T = e\ in\ t_1\rangle) \between \acute{C}_2 & & \\ (\nu(\Sigma).(\nu(\vec{n}_1{:}\vec{T}_1).\ C_1' \parallel n\langle t_1\rangle) \between \acute{C}_2 & \cdots\cdots & \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle\ \ \ \ \ \ \ \ \ \ t\rangle) \end{array}$$

Proof. By induction on the definition of $\between$.
Case: $C_1 \between C_2 = C_1 \between \mathbf{0} \equiv C_1 = \nu(\vec{n}{:}\vec{T}).\ (C_1' \parallel n\langle let\ x{:}T = e\ in\ t\rangle)$
In this case, s and $\bar{s}$ are empty, $\acute{\Sigma} = \Sigma$, $\vec{n}_1 = \vec{n}$, and $t_1 = t$.
Case: $C_1 \between C_2 = (\nu(n{:}T).\tilde{C}_1) \between C_2 \equiv \nu(n{:}T).(C_1 \between C_2)$,
where $n \notin fn(C_2)$.
Case: $C_1 \between C_2 \equiv \between$

□

Lemma 18 (Decomposition and $\rightsquigarrow$-step). *If* $C_1 \between C_2 \equiv C$ *and* $C \rightsquigarrow \acute{C}$, *then there exists a trace* s *such that* $\Delta, \Phi; E_\Delta \vdash C_1 : \Theta, \Psi; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}, \Phi; \acute{E}_\Delta \vdash \acute{C}_1 : \acute{\Theta}, \acute{\Psi}; \acute{E}_\Theta$ *and* $\Theta, \Phi; E_\Theta \vdash C_2 : \Delta, \Psi; E_\Delta \stackrel{\bar{s}}{\Longrightarrow} \acute{\Theta}, \Phi; \acute{E}_\Theta \vdash \acute{C}_2 : \acute{\Delta}, \acute{\Psi}; \acute{E}_\Delta$, *where* $\nu(\acute{\Sigma} \setminus \Sigma).\ \acute{C}_1 \between \acute{C}_2 \equiv \acute{C}$ *and where* $\Sigma = \Delta, \Theta, \Phi$ *and* $\acute{\Sigma} = \acute{\Delta}, \acute{\Theta}, \acute{\Phi}$. *Pictorially:*

$$\begin{array}{ccc} C_1 \between C_2 & \equiv\equiv\equiv & C \\ \Big\downarrow{\scriptstyle \bar{s}\,s} & & \Big\downarrow \\ \nu(\acute{\Sigma} \setminus \Sigma).\ (\acute{C}_1 \between \acute{C}_2) & \cdots\cdots & \acute{C} \end{array}$$

Proof. We start by looking at the form of the component C. It is able to do an immediate $\rightsquigarrow$-step, which must (ultimately) be justified by one of the axioms Red, ... , NewT from Table 5, or by NewO_{lazy} from Table **??**. This means, a thread in C executes an top-most let-command, i.e.,

$$C = \nu(\vec{n}{:}\vec{T}).\ (C' \parallel n\langle let\ x{:}T = e\ in\ t\rangle)$$

□

Lemma 19 (Decomposition and τ-step).
Proof. □

Lemma 20 (Trace decomposition). *Assume* $\Delta, \Phi \vdash C_1 : \Theta, \Psi$ *and* $\Theta, \Phi \vdash C_2 : \Delta, \Psi$ *with* $C_1 \between C_2 \equiv C$. *If* $C \Longrightarrow C'$, *then* $\Delta, \Phi \vdash C_1 : \Theta, \Psi \stackrel{s}{\Longrightarrow} \Delta', \Phi \vdash C_1 : \Theta', \Psi'$ *and* $\Theta, \Phi \vdash C_2 : \Delta, \Psi \stackrel{\bar{s}}{\Longrightarrow} \Theta', \Phi \vdash C_2' : \Delta', \Psi'$, *for some trace* s *where* $C' \equiv \nu(\Delta', \Theta', \Psi' \setminus \Delta, \Theta, \Psi).\ C_1' \between C_2'$.

Proof (of trace decomposition (Lemma 20)). **status:** not done.

A.3 Legal traces

In this section we prove the results about the legal traces. The most important is that the actually the observable behavior of a well-typed component.

We start with the auxiliary definition concerning the parenthetic nature of calls and returns of a legal thread. As in [JR02], we define *balance* of a thread and the operation *pop* as follows:

Definition 14 (Balance, Pop). *The thread n is* balanced *in a sequence s if one of the following conditions holds:*

1. *If $n \notin thread(s)$, then n is balanced in s.*
2. *If n is balanced in s_1 and s_2, then it is balanced in $s_1 s_2$.*
3. *n is balanced in $\nu(\Delta, \Theta).n\langle call\ o_2.l(\vec{v})\rangle?\ s\ \nu(\Theta', \Delta').n\langle return(v)\rangle!$, if it is balanced in s.*
4. *n is balanced in $\nu(\Theta, \Delta).n\langle call\ o_2.l(\vec{v})\rangle!\ s\ \nu(\Delta', \Theta').n\langle return(v)\rangle?$, if it is balanced in s,*

The function pop is defined as follows:

1. *$pop\, n\ s = \bot$, if n is balanced in s*
2. *$pop\, n\ (s_1 a s_2) = a$ if $a = \nu(\Delta, \Theta).\ n\langle call\ o_2.l(\vec{v})\rangle?$ and n is balanced in s_2.*
3. *$pop\, n\ (s_1 a s_2) = a$ if $a = \nu(\Delta, \Theta).\ n\langle call\ o_2.l(\vec{v})\rangle!$ and n is balanced in s_2.*

With this definition, we can define when a thread can perform a input or output action in the next step. Input enabledness stipulates whether, given a sequence of past communication labels, an incoming call is possible in the next step; analogously for output enabledness. Note that enabledness in the definition refers to calls, only, not to returns; they are checked directly. To be input enabled, it suffices to check that the thread has *left* the component, or that it is fresh to the component.

Definition 15 (Enabledness). *Let γ be a method call of the form $\nu(\Delta, \Theta).n\langle call\ o_2.l(\vec{v})\rangle$. Then call-*enabledness *of γ after the history r and in the contexts Δ and Θ is defined as:*

$$\Delta; E_\Delta \vdash r \rhd \gamma? : \Theta; E_\Theta \text{ if } (n \notin dom(\Theta) \text{ or } n \in dom(E_\Theta)) \tag{21}$$

$$\Delta; E_\Delta \vdash r \rhd \gamma! : \Theta; E_\Theta \text{ if } (n \notin dom(\Delta) \text{ or } n \in dom(E_\Delta)) \tag{22}$$

We also say, a thread is *input-call enabled* after r if $\Delta \vdash r \rhd \gamma? : \Theta$ for some incoming call label and we call the condition $pop\, n\ r = \nu(\Delta').\ n\langle call\ o_2.l(\vec{v})\rangle!$ also *input-return* enabledness. The definitions are used dually for output-call enabledness and output-return enabledness. When leaving the kind of communication unspecified we just speak of input-enabledness or output-enabledness. Note that return-enabledness implies call-enabledness, but not vice versa.

The following lemma states a simple "invariant" about the form of the graphs encoded by E_Δ and E_Θ for legal traces. The two environments E_Δ and E_Θ connect object and thread names. The use of thread names is restricted, however:

All known thread names are acquainted to exactly *one* object name, and the connection is justified either by the assumption contect E_Δ or the commitment context E_Θ. No thread name ever is ever known by another named entity. This reflects the intuitition, that the top-most frame of each thread has to remember the identity of the *caller* object.[32] The Lemma is the analogue to Lemma 7 for the dynamic semantics.

Lemma 21 (Invariants). *Derivations for legal traces for* $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\ \Theta; E_\Theta$ *preserve the following invariants for all subgoals* $\Delta'; E'_\Delta \vdash r' \rhd s' : \mathit{trace}\ \Theta; E'_\Theta$*:*

1. $E'_\Delta \subseteq \Delta' \times (\Delta' + \Theta')$ *and* $E'_\Theta \subseteq \Theta' \times (\Theta' + \Delta')$.
2. $dom(\Delta') \cap dom(\Theta') = \emptyset$*, for all object and class references.*
3. *for all thread names* n*:*
   (a) *either* $n \notin \Delta' \cup \Theta'$ *or else* $n \in \Delta' \cap \Theta'$.
   (b) *if* $n \in \Delta' \cap \Theta'$*, then*
      i. *either* $n \hookrightarrow o \in E'_\Delta$*, for some object name* $o \in \Theta'$*,*
      ii. *or else* $n \hookrightarrow o \in E'_\Theta$*, for some object name* $o \in \Delta'$.
   (c) *there exists no name* p *such that* $p \hookrightarrow n \in E'_\Delta$ *or* $p \hookrightarrow n \in E'_\Theta$

Proof. By induction on the rules from Table 13. □

Lemma 22 (Enabledness). *Assume* $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\ \Theta; E_\Theta$.

1. *If* n *is input-return enabled in* r*, then it is input-call enabled in* r*, and dually for output-return enabledness and output-call enabledness.*
2. *For each thread identifier* $n \in \Delta$*, either* n *is input-enabled or* n *is output-enabled after* r.

Proof. Part (1) is trivial as return enabledness is directly defined as special case call enabledness. For part (2), assume n is input-enabled, which by the first part implies that n is call-enabled. Then the result follows by Lemma 21, especially part 3b. □

The following reformulates enabledness of a thread n exploiting the invariants mentioned before.

Lemma 23. *Assume* $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\ \Theta; E_\Theta$*. Then for each thread identifier* $n \in \Delta$ *we have:*

1. n *is output enabled after* r *iff* $\Delta; E_\Delta \vdash n \hookrightarrow o : \Theta$ *or* $n \notin \Delta \cup \Theta$.
2. n *is input enabled after* r *iff* $\Theta; E_\Theta \vdash n \hookrightarrow o : \Delta$ *or* $n \notin \Delta \cup \Theta$.

Proof. Directly from the definition of enabledness (Definition **??**) and using the properties of Lemma 21. □

The following easy lemma characterizes the change of enabledness when a trace is extended by a communication label.

[32] Remember that the identity of the callee, which replaces the self-parameter of a method need to be remembered for the purpose of characterizing the legal trace.

Lemma 24 (Enabledness). *Assume a thread name n and a legal trace $\Delta; E_\Delta \vdash r : \Theta; E_\Theta$ according to Table 13.*

1. *If a be an input-call label. If n is input-call enabled after r, then it is output-return enabled after $r\ a$.*
2. *If a is an input-return label and if n is input-return enabled after r, then n is output enabled after $r\ a$.*

The situation in both cases is dual for output labels.

Proof. Cf. Definition **??** and Definition 14. Both parts of the Lemma follow directly from the definition of enabledness. Especially, $pop\ n\ (r\ a) = a$ is a direct consequence of the definition of *pop* and balance. □

Lemma 25.

Lemma 26 (Prefix). *If $\Delta; E_\Delta \vdash st : trace\,\Theta; E_\Theta$, then $\Delta; E_\Delta \vdash s : trace\,\Theta; E_\Theta$ by a proper sub-derivation, if t is not empty.*

Proof. Immediate by the rules of Table **??**: each label corresponds to an instance of one rule. □

The following lemmas states that the type system for legal traces yields a safe or sound overapproximation of the actual behavior of the labeled transition system.

Lemma 27 (Soundness of legal traces). *If $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ and $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow}$, then $\Delta; E_\Delta \vdash s : trace\,\Theta; E_\Delta$.*

status: Not yet done.

Proof. □

Lemma 28 (Trace duality). *If $\Delta; E_\Delta \vdash s : trace\,\Theta; E_\Theta$, then $\Theta, E_\Theta \vdash \bar{s} : trace\,\Delta; E_\Delta$.*

status: Seems clear.

Proof. By induction on the length of derivation of $\Delta; E_\Delta \vdash s : trace\,\Theta; E_\Theta$. Each rule of Table 13 has a dual counterpart. □

A.4 Closure

In this section we prove a few simple results about the closure relation. As in the sequential setting, a component is internally deterministic, the closure set has a rather simple form: the set of traces is closed under prefixing, and in a situation where the component is input enabled, the trace can be extended by an incoming communication.

The following lemma specifically holds only in the single-threaded case.

Lemma 29. *If $\Delta; E_\Delta \vdash s_1\gamma! \sqsubseteq s_2\gamma! : trace\,\Theta; E_\Theta$ and $\gamma! \notin s_1, s_2$, then $s_1 = s_2$.*

Proof. Immediate from the definition of $\sqsubseteq$ and from the assumption that the output label $\gamma!$ occurs neither in s_1 nor in s_2. □

Lemma 30 (Information order duality). *If* $\Delta; E_\Delta \vdash s_1\gamma! \sqsubseteq s_2\gamma! : \mathit{trace}\,\Theta; E_\Theta$ *and* $\gamma! \notin s_1, s_2$*, then* $\Theta; E_\Theta \vdash \bar{s}_2 \sqsubseteq \bar{s}_1 : \mathit{trace}\,\Delta; E_\Delta$*.*

Proof. Immediate from reflexivity of $\sqsubseteq$, trace duality, and Lemma 29. □

The following lemmas for information order closure justifies the definition the $\sqsubseteq$-relation: If a component realizes a trace s, all traces in the closure, i.e., all traces $\sqsubseteq s$, are also possible.

Lemma 31 (Information order closure). *If* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s_2}{\Longrightarrow}$ *and* $\Delta; E_\Delta \vdash s_1 \sqsubseteq s_2 : \mathit{trace}\,\Theta; E_\Theta$*, then* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s_1}{\Longrightarrow}$*.*

status: seems ok.

Proof. Proceed by induction on the length of the derivation for the statement $\Delta; E_\Delta \vdash s_1 \sqsubseteq s_2 : \mathit{trace}\,\Theta; E_\Theta$. The cases for reflexivity, transitivity, and prefixing are trivial.

Case: O-Input: $\Delta; E_\Delta \vdash s\gamma? \sqsubseteq s : \mathit{trace}\,\Theta; E_\Theta$
So we are given $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$. According to assumption, $s\gamma?$ is a legal trace i.e., $\Delta; E_\Delta \vdash s\gamma? : \mathit{trace}\,\Theta; E_\Theta$, and we distinguish the two subcases:

Subcase: $\gamma? = n\langle \mathit{call}\ o_2.l(\vec{v})\rangle?$
In this case, legality is justified by rule L-CallIn in the last step. Inverting this rule yields that the component is input enabled, i.e., $\Delta \vdash s \rhd n\langle \mathit{call}\ o_2.\vec{v}\rangle?$, and furthermore that the incoming values meet the assumptions.[33] By definition of enabledness (Definition **??**), either $n \notin \Delta$ and $\mathit{pop}\ \ n\ s = \bot$, or $\mathit{pop}\ \ n\ s = \gamma'!$ which implies using the two subcases of Lemma 22(**??**) that either $n \notin \mathit{dom}(\Theta)$ or $\acute{C} = C' \parallel n\langle \mathit{let}\, x{:}T' = o_2\ \mathit{blocks\ for}\ o_1\ \mathit{in}\ t\rangle$. According to the situation, the component $\acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$ accepts the incoming call by rule CallIn$_1$ respectively CallIn$_2$, and instances of rule BIn.

Subcase: $\gamma? = n\langle \mathit{return}(v)\rangle?$
Inverting rule L-ReturnIn gives that $\mathit{pop}\, ns = \nu(\Theta').n\langle \mathit{call}\ o_2.l(\vec{v})\rangle!$, hence by Lemma 22(**??**), $\acute{C}$ is of the form $C' \parallel n\langle \mathit{let}\, x{:}T = o_2\ \mathit{blocks\ for}\ o_1\ \mathit{in}\ t\rangle$, hence the step can be taken by rule ReturnIn and instances of rule BIn.

□

Corollary 1 (Subsumption). *If* $\Delta; E_\Delta \vdash s_2 : \mathit{trace}\,\Theta; E_\Theta$ *and* $\Delta; E_\Delta \vdash s_1 \sqsubseteq s_2 : \Theta; E_\Theta$*, then* $\Delta; E_\Delta \vdash s_1 : \mathit{trace}\,\Theta; E_\Theta$*.*

status: seem ok

Proof. By definability (Theorem 1), there exists a component C_{s_2} such that $\Delta; E_\Delta \vdash C_{s_2} : \Theta; E_\Theta \stackrel{s_2}{\Longrightarrow}$. By information order closure from Lemma 31 above, also $\Delta; E_\Delta \vdash C_{s_1} : \Theta; E_\Theta \stackrel{s_1}{\Longrightarrow}$, whence the result follows by soundness of traces from Lemma 27.

Note that the argument did not use the full power of definability of Theorem 1: it suffices that there *exists* a component C_{s_2} realizing s_2 and all traces in its closure. The inverse direction that C_{s_1} does not realize more than those traces is not needed. □

[33] We ignore the static system here.

A.5 Definability

A.5.1 Synchronization code A core construction we need is to broadcast connectivity information through the object graph. An important invariant is that the objects forming a clique are *fully connected* and reflect the specified connectivity of the commitment context E_Θ. The method of Definition 16 is responsible for broadcasting any newly learnt names across a clique, thereby re-establishing the full connectivity of the clique (cf. Lemma **??**. The clique broadcast is done by a method $propagate : \vec{c} \to \mathsf{Unit}$. It takes an array of (object) names, compares it with the ones the object already knows. If this knowledge is up-to date, the work is done. Otherwise, the object updates its information and informs all objects it knows about the newly learned connectivity, which will propagate them further.

Definition 16 (Propagate). *Each object supports a method propagate implemented as:*

$$\begin{array}{lll} propagate(\vec{o} : \vec{c}) \triangleq & let\ \vec{o}_{new} = self.\Theta - \vec{o}\ \ in & \\ & \mathsf{if} \quad is\_empty(\vec{o}_{new}) & \\ & \mathsf{then}\ () & \\ & \mathsf{else}\ \ self.\Theta := self.\Theta + \vec{o}_{new}; & \\ & \quad \forall o' \in \Theta.\ o'.propagate(self.\Theta)\ . & \end{array} \tag{23}$$

Note that in the recursive self-call of the method, all known objects, i.e., the value of the instance variable are handed over, not just the newly learnt objects.

The next lemmas characterize the behavior of the propagate method. First the one as needed in the synchronization code for input. In that situation, the external input step hands over the argument references by substitution. At this point, before they are stored in instance variables (done by *propagate*, btw.), the component is still fully connected, but since the commitment contexts Θ and E_Θ are already extended to reflect the new information, the component at this very point does no longer satisfy the commitment in the strict way (i.e., using $\vdash_{\mathcal{S}}$). The propagation procedure stores the newly learnt connectivity information in the target object of the communication thereby violating the invariant of full-connectivity; upon return, full-connectivity is reestablished:[34]

Lemma 32 (Propagate: Input). *Assume a fully-connected component* $\Delta; E_\Delta \vdash_{\mathcal{S}} C' \parallel n\langle o.propagate(\vec{o}); t\rangle : \Theta; E_\Theta$.

$$\begin{array}{l} \Delta; E_\Delta \vdash_{\mathcal{S}} C' \parallel n\langle o.propagate(\vec{o}); t\rangle : \Theta; E_\Theta \Longrightarrow \\ \Delta; E_\Delta \vdash_{\mathcal{S}} C'' \parallel n\langle t\rangle : \Theta, E_\Theta + (o \hookrightarrow \vec{o}, n \hookrightarrow [o]) \end{array}$$

Proof (of the propagation Lemma 32). In absence of interference, the propagate method realizes a standard depth-first graph traversal of the objects of the clique.[35] □

[34] As the lemma is formulated to describe the situation immediately after an incoming communication, the thread n is already visible to the outside.

[35] Since the graph structure of the clique is rather simple —the objects are fully connected, with the potential exception of the argument object references. This means

Next a similar property, now tailored to the situation for output communication. In that case, the task of the component is to create any required new objects to be made known to the environment, and it uses the *propagate*-method to make the new objects known internally before communicating their identities.

Lemma 33 (Propagate: Output). *Assume a well-typed, fully-connected component* $\Delta; E_\Delta \vdash_{\mathcal{S}} C : \Theta; E_\Theta$ *of the form*

$$\Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\vec{n}{:}\vec{T}, \Theta', \Delta').(C' \parallel n\langle o.propagate(\Theta', \Delta'); t\rangle) : \Theta; E_\Theta \ ,$$

where Θ' *and* Δ' *are object names bindings with the usual conventions, i.e.* $\Theta' = \vec{o}{:}\vec{c}$ *and* $\Delta' = \vec{o'}{:}\vec{c'}$ *where* $\Theta \vdash c_i : [\![T_i]\!]$ *and* $\Delta \vdash c'_i : [\![T_i]\!]$ *for all* c_i *and* c'_j*. Assume further* $\Theta; E_\Theta \vdash n \hookrightarrow o : \Delta$*, i.e., the thread* n *is currently visiting* o*. Then*

$$\Delta; E_\Delta \vdash \nu(\vec{n}{:}\vec{T}, \Theta', \Delta').(C' \parallel n\langle o.propagate(\Theta', \Delta'); t\rangle) : \Theta; E_\Theta \implies$$
$$\Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\vec{n}{:}\vec{T}, \Theta', \Delta').(C'' \parallel n\langle t\rangle) : \Theta; E_\Theta \ .$$

Proof (of the propagation Lemma 33). Analogous to the proof of the propagation Lemma 32 for input. □

The following corollary combines the previous lemma with the creation of objects from within the component and thus describes the situation for outgoing communication, i.e., in particular the situation for Lemma 1

Corollary 2 (Propagate).

$$\Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\vec{n}{:}\vec{T}).C' \parallel n\langle let\ \vec{x}{:}\vec{c} = o.\ new(\vec{c})\ in\ o.propagate(\vec{x}); t\rangle : \Theta; E_\Theta \implies$$
$$\Delta; E_\Delta \vdash_{\mathcal{S}} \nu(\vec{n}{:}\vec{T}, \Theta', \Delta').C'' \parallel n\langle t\rangle : \Theta; E_\Theta$$

Proof. A direct consequence of the propagation Lemma **??** and the rule from Table 5 resp. Table **??**, in particular NEW$_i$ and NEWO$_{lazy}$. □

---

the structure of the graph is not very "deep" and depth-first traversal seems weird at first sight. For instance, if a clique simply learns one new fresh identity one could do without recursion just instructing each object within reach, i.e., at a distance of one, to update its knowledge. The situation is not *that* simple, however. In the case when propagation is used to *merge* two or more cliques, the information of the previously disjoint cliques has to be combined and broadcast to all members. However, one could think of more efficient algorithm exploiting better the fact that we are dealing with groups of fully-connected groups of objects, but efficiency is not our current concern.

Proof (of Theorem 1 (Definability)). There are two directions to show. We start with the easier "if"-direction.

So assume a legal trace $\Delta; E_\Delta \vdash s : \mathit{trace}\,\Theta; E_\Theta$ and assume $\Delta; E_\Delta \vdash s' \sqsubseteq s\ \mathit{trace} : \Theta; E_\Theta$. Then show that $\Delta; E_\Delta \vdash C_s : \Theta; E_\Theta \overset{s'}{\Longrightarrow}$. With Lemma 31 it suffices to show that $\Delta; E_\Delta \vdash C_s : \Theta; E_\Theta \overset{s}{\Longrightarrow}$. We prove this by induction on the definition of C_s, which ultimately means by induction on the length of the legality derivation for s (cf. the Definition 2).

Case: L-Empty
Immediate, as the empty trace is always possible.

Case: L-CallOut
We are given $s = as'$, where $a = \nu(\Theta', \Delta').\ n\langle \mathit{call}\ o_2.l(\vec{v})\rangle!$. By construction, $C_{as'}$ is of the form (cf. Equation 14):

$$\vdash C_{s'} \parallel n\langle \mathit{create}(\Theta'); \mathit{propagate}(\Theta'); \mathit{wait}(o_2, \vec{v}); o_2.\mathit{delegate}_l(o_1, \vec{v}); t\rangle$$

By the rules of the operational semantics and Lemma **??**

$$C_{as'} \rightsquigarrow^* \equiv \nu(\Theta').C_{s'} \parallel n\langle \mathit{propagate}(\Theta'); \mathit{wait}(o_2, \vec{v}); o_2.\mathit{delegate}_l(o_1, \vec{v}); t\rangle$$

□

Data structures

status: not done, but with all other lemmas in place, it's simple.

Proof (of completeness (Theorem 2)). □

Index

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

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\vec{v})\rangle? \quad \Delta \vdash \epsilon \rhd a : \Theta \quad static(\Delta, \Theta) \quad \Delta \vdash \odot \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; \odot \hookrightarrow (\Delta', \Theta') \setminus n \\ ; \acute{\Theta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!] \quad \acute{\Delta} \vdash n{:}\, thread \quad \acute{\Theta} \vdash n{:}\, thread \quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \qquad \acute{\Delta}; \acute{E}_\Delta \vdash a_\odot \rhd s : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \epsilon \rhd a\ s : trace\,\Theta; E_\Theta}\ \text{L-CalliI}_i$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle call\ o_2.l(\vec{v})\rangle! \quad \Delta \vdash \epsilon \rhd a : \Theta \quad static(\Theta, \Delta) \quad \Theta \vdash \odot \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + (\Delta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; \odot \hookrightarrow (\Theta', \Delta') \setminus n \\ ; \acute{\Theta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!] \quad \acute{\Delta} \vdash n{:}\, thread \quad \acute{\Theta} \vdash n{:}\, thread \quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ dom(\Theta', \Delta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \qquad \acute{\Delta}; \acute{E}_\Delta \vdash a_\odot \rhd s : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \epsilon \rhd a\ s : trace\,\Theta; E_\Theta}\ \text{L-CalloO}_i$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle call\ o_2.l(\vec{v})\rangle? \qquad \Delta \vdash o_1 : c_1 \qquad \Delta \vdash r \rhd a : \Theta \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \setminus n \\ ; \acute{\Theta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!] \quad \acute{\Delta} \vdash n{:}\, thread \quad \acute{\Theta} \vdash n{:}\, thread \quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ dom(\Delta', \Theta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Delta}; \acute{E}_\Delta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Theta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a_{o_1} \rhd s : trace\,\acute{\Theta}; \acute{E_\Theta} \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\,\Theta; E_\Theta}\ \text{L-CalliI}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle return(v)\rangle! \qquad pop\,n\ r = \nu(\Delta'', \Theta'').\ n\langle [o_1] call\ o_2.l(\vec{v})\rangle? \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow v, n \hookrightarrow o_1 \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow (\Theta', \Delta') \setminus n \\ dom(\Theta', \Delta') \subseteq fn(v) \qquad \Theta \vdash o_2 : [\dots, l{:}\vec{T} \to T, \dots] \qquad ; \Delta, \acute{\Theta} \vdash v : T \\ \acute{\Theta}; \acute{E}_\Theta \vdash o_2 \leftrightharpoons\!\hookrightarrow v : \acute{\Delta} \qquad \acute{E}_\Delta; \acute{\Delta} \vdash r\ a \rhd s : trace\,\acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\,\Theta; E_\Theta}\ \text{L-RetO}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ n\langle call\ o_2.l(\vec{v})\rangle! \qquad \Theta \vdash o_1 : c_1 \qquad \Delta \vdash r \rhd a : \Theta \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2 \quad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_1 \hookrightarrow (\Theta', \Delta') \setminus n \\ ; \acute{\Delta} \vdash o_2 :: [\![\dots, l{:}\vec{T} \to T, \dots]\!] \quad \acute{\Delta} \vdash n{:}\, thread \quad \acute{\Theta} \vdash n{:}\, thread \quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ dom(\Theta', \Delta') \subseteq fn(n\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Theta}; \acute{E}_\Theta \vdash n \leftrightharpoons o_1 \leftrightharpoons\!\hookrightarrow \vec{v}, o_2 : \acute{\Delta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a_{o_1} \rhd s : trace\,\acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\,\Theta; E_\Theta}\ \text{L-CalloO}$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ n\langle return(v)\rangle? \quad pop\,n\ r = \nu(\Theta'', \Delta'').\ n\langle [o_1] call\ o_2.l(\vec{v})\rangle! \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + (\Theta'; o_1 \hookrightarrow v, n \hookrightarrow o_1) \quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow (\Delta', \Theta') \setminus n \\ dom(\Delta', \Theta') \subseteq fn(v) \qquad \Delta \vdash o_2 : [\dots, l{:}\vec{T} \to T, \dots] \qquad ; \acute{\Delta}, \acute{\Theta} \vdash v : T \\ \acute{\Delta}; \acute{E}_\Delta \vdash o_2 \leftrightharpoons\!\hookrightarrow v : \acute{\Theta} \qquad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : trace\,\acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\,\Theta; E_\Theta}\ \text{L-RetI}$$

Table 13. Legal traces

$$\Delta; E_\Delta \vdash s \sqsubseteq sr : \mathit{trace}\,\Theta; E_\Theta \qquad \text{O-Pref}$$

$$\Delta; E_\Delta \vdash s\gamma? \sqsubseteq s : \mathit{trace}\,\Theta; E_\Theta \qquad \text{O-Input}$$

$$\frac{\text{replay}}{\Delta; E_\Delta \vdash s\gamma! \sqsubseteq s : \mathit{trace}\,\Theta; E_\Theta}\ \text{O-Output}$$

$$\Delta; E_\Delta \vdash s\gamma_2?\gamma_1!r \sqsubseteq s\gamma_1!\gamma_2?r : \mathit{trace}\,\Theta; E_\Theta \qquad \text{O-OI}$$

$$\frac{\Delta; E_\Delta \vdash s \rhd \mathit{receiver}(\gamma_1?) \not\equiv \mathit{sender}(\gamma_2!) : \Theta; E_\Theta}{\Delta; E_\Delta \vdash s\gamma_2!\gamma_1?r \sqsubseteq s\gamma_1?\gamma_2!r : \mathit{trace}\,\Theta; E_\Theta}\ \text{O-IO}$$

$$\Delta; E_\Delta \vdash s\ \nu(\Delta').\ \gamma_2?\ \gamma_1?\ r \sqsubseteq s\ \nu(\Delta').\ \gamma_1?\ \gamma_2?\ r : \mathit{trace}\,\Theta; E_\Theta \qquad \text{O-II}$$

$$\Delta; E_\Delta \vdash s\ \nu(\Theta').\ \gamma_2!\ \gamma_1!\ r \sqsubseteq s\ \nu(\Theta').\ \gamma_1!\ \gamma_2!\ r : \mathit{trace}\,\Theta; E_\Theta \qquad \text{O-OO}$$

Table 14. Information preorder

$$\frac{}{\Delta; E_\Delta \vdash \mathbf{0} : (); ()}\ \text{T-Empty}$$

$$\frac{\Delta, \Theta_2; E_\Delta, E_{\Theta_2} \vdash C_1 : \Theta_1; E_{\Theta_1} \qquad \Delta, \Theta_1\ E_\Delta, E_{\Theta_2} \vdash C_2 : \Theta_2; E_{\Theta_2}}{\Delta; E_\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2; E_{\Theta_1}, E_{\Theta_2}}\ \text{T-Par}$$

$$\frac{\Delta; E_\Delta \vdash C : \Theta, n{:}T; E_\Theta + n}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta}\ \text{T-Nu}$$

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

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

$$\frac{\Delta; E_\Delta \vdash \Delta'; E'_\Delta \qquad \Delta'; E'_\Delta \vdash C : \Theta'; E'_\Theta \qquad \Theta'; E'_\Theta \vdash \Theta; E_\Theta}{\Delta; E_\Delta \vdash C : \Theta; E_\Theta}\ \text{T-Weaken}$$

Table 15. Static semantics with connectivity (configurations)

$$
\begin{array}{rcl}
t_{ire} & ::= & t^i_{body}; t_{ie} \\
t_{ie} & ::= & t_{ire} \\
& | & \epsilon \\
t^i_{body} & ::= & \mathit{let}\ \ y{:}T' = o_1\ \mathit{blocks\ for}\ o_2 \\
& & \mathit{in}\ \ (\mathit{let}\, x{:}T = t^i_{sync}(y, \check{a}); t_{oe}; t^o_{sync}; v\, \mathit{in}\, o_2\ \mathit{return\ to}\ o_1\ x) \\
t^i_{body} & ::= & \mathit{let}\ \ y{:}T' = o_1\ \mathit{blocks\ for}\ o_2 \\
& & \mathit{in}\ \ t^i_{sync}(y); (\mathit{let}\, x{:}T = t_{oe}; t^o_{sync}; v\, \mathit{in}\, o_2\ \mathit{return\ to}\ o_1\ x) \\
t_{ore} & ::= & t^o_{body}; t_{ie} \\
t^o_{body} & ::= & \mathit{let}\, x{:}T = t_{oe}; t^o_{sync}; v\, \mathit{in}\, o_2\ \mathit{return\ to}\ o_1\ x \\
& | & \mathit{stop} \\
t_{oe} & ::= & t^o_{sync}; (\mathit{let}\, y{:}T = o.l(\vec{v})\, \mathit{in}\, t^i_{sync}(y)); t_{oe} \\
& | & \epsilon
\end{array}
$$

Table 16. Input and output enabled threads

$$
\begin{array}{rcll}
\mathbf{0} \between C & \equiv & C & \\
(\nu(n{:}T).C_1) \between C_2 & \equiv & \nu(n{:}T).(C_1 \between C_2) & n \notin \mathit{fn}(C_2) \\
(o[O] \parallel C_1) \between C_2 & \equiv & o[O] \parallel (C_1 \between C_2) & \\
(c[\![O]\!] \parallel C_1) \between C_2 & \equiv & c[\![O]\!] \parallel (C_1 \between C_2) & \\
(n\langle t\rangle \parallel C_1) \between C_2 & \equiv & n\langle t\rangle \parallel (C_1 \between C_2) & n \notin \mathit{dom}(C_2) \\
(n\langle t_1\rangle \parallel C_1) \between (n\langle t_2\rangle \parallel C_2) & \equiv & n\langle t_1 \between t_2\rangle \parallel (C_1 \between C_2) & n \notin \mathit{dom}(C_2)
\end{array}
$$

$$
\begin{array}{rcl}
(\mathit{let}\, x{:}T = \ o_1\ \mathit{blocks\ for}\ o_2\, \mathit{in}\, t) \between \mathit{stop} & \equiv & \mathit{stop} \\
(\mathit{let}\, x{:}T = \ o_1\ \mathit{blocks\ for}\ o_2\, \mathit{in}\, t) \between v & \equiv & v \\
(\mathit{let}\, x{:}T = o_1\ \mathit{blocks\ for}\ o_2\, \mathit{in}\, t_1) \between & & \\
(\mathit{let}\, y{:}T' = o_2\ \mathit{return\ to}\ o_1\ v; t'_2) & \equiv & t_1[v/x] \between t'_2 \\
(\mathit{let}\, x{:}T = o_1\ \mathit{blocks\ for}\ o_2\, \mathit{in}\, t_1) \between (\mathit{let}\, y{:}T' = e\, \mathit{in}\, t_2) & \equiv & \mathit{let}\, y{:}T' = e\, \mathit{in} \\
& & ((\mathit{let}\, x{:}T = o_1\ \mathit{blocks\ for}\ o_2\, \mathit{in}\, t_1) \between t_2)
\end{array}
$$

Table 17. Merge