Observability, Connectivity, and Replay in a Sequential Calculus of Classes*

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

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

Abstract. Object calculi have been investigated as semantical foundation for object-oriented languages. Often, they are object-based, whereas the mainstream of object-oriented languages is *class-based.*
Considering classes as part of a component makes instantiation a possible interaction between component and environment. As a consequence, one needs to take *connectivity* information into account.
We formulate an operational semantics that incorporates the connectivity information into the scoping mechanism of the calculus. Furthermore, we formalize a notion of equivalence on traces which captures the uncertainty of observation cause by the fact that the observer may fall into separate groups of objects. We use a corresponding trace semantics for full abstraction wrt. a simple notion of observability. This requires to capture the notion of *determinism* for traces where classes may be instantiated into more than one instance during a run and showing thus twice an equivalent behavior (doing a "replay"), a problem absent in an object-based setting.
Keywords: class-based object-oriented languages, formal semantics, determinism, full abstraction

1 Introduction

Classes are a structuring concept for object-oriented languages such as *Java* or $C^{\#}$. This raises the question what the semantics of a program is when considering classes as composition units. A simple, elegant, and common semantical approach is to take an observational point of view: two program fragments are equal, if, when put in any possible context, no difference can be seen. Starting from a simple notion of observation, [10] presented a fully abstract trace semantics for a multithreaded *object* calculus, i.e., a language without classes; [2] generalized the result by taking *classes* into account.

In this paper, we re-address the problem in a *single-threaded* setting. This is interesting for two reasons. First, simplifying the language does not simplify

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

the problem *per se.* Certain complications in connection with concurrency certainly get simpler, e.g., by absence of race conditions. On the other hand, new complications arise. In particular, with one thread only, the language becomes *deterministic* which needs to be accounted for in the description of the semantics. Secondly, concentrating on a single thread allows to understand the semantical impact of classes more clearly and independently from the orthogonal aspects of concurrency.

One key observation is that in the presence of classes one needs to take connectivity information into account, i.e., the way objects may have knowledge of each other, to characterize the observable behavior. In particular, unconnected environment objects can neither determine the absolute order of interaction, nor can they exchange information to compare object identities. Furthermore, with a deterministic language, one needs to capture the notion of *determinism* for traces where classes may be instantiated into more than one instance during a run and showing thus twice an equivalent behavior (doing a "replay"), a problem absent in an object-based setting.

Overview The paper is organized as follows. We start in Section 2 with an informal account of the semantics and the underlying intuitions. Section 3 contains the syntax of the calculus and a sketch of its semantics. In particular, the notions of lazy instantiation and connectivity of objects are formalized. Afterwards, Section 4 elaborates on the trace semantics and in particular an equivalence relation on traces capturing the uncertainty of observation in a class-based setting. In Section 5 we fix the notion of observation and state the full abstraction result. Section 6 concludes with related and future work.

2 Observability and classes

This section presents on an intuitive level the consequences of incorporating *classes* into the observational set-up.

2.1 Cross-border instantiation and connectivity

The observational set-up separates *classes* into component and environment classes. Hence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well.

If, for instance, the component creates an instance of an environment class, the interaction between the component and the newly created object can entail observable effects in the future, as the code of the object is externally provided and therefore this interaction belongs to the externally visible observer-program behavior. Hence, instances of environment classes belong to the environment, and dually those of internal classes to the component. However, in the above situation, the *reference* to the new external object is kept at the creator for the time being. So if the component instantiates two objects o_2 and o_3 of the environment, the situation looks informally as in Figure 1, where the dotted bubbles indicate the scope of o_2, respectively of o_3.

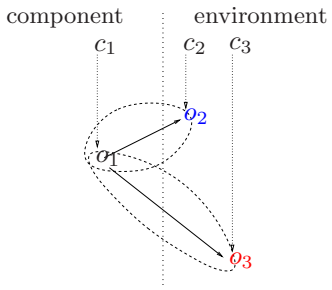


Fig. 1. Instances of external classes

For an exact account of the semantics, the inability of o_2 and o_3 to be in contact must be accounted for. More generally, the semantics must contain a representation of which object can possibly be in contact with others, i.e., an overapproximation of the heap's *connectivity*. Sets of objects which can possibly be in contact with each other form therefore equivalence classes of names —we call them *cliques*— and 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.

2.2 Different observers and order of events

That the observer may fall into separate cliques of unconnected objects has implications for what can be observed. First of all, the absolute *order of events* cannot be determined, as the observer cliques may not be able to coordinate. For instance, the environment or observer, split into 1 and 2 on the right-hand sides of the three scenarios of Figure 2 cannot distinguish between the three variants of the component on the respective left-hand sides. Note that the clique struc-

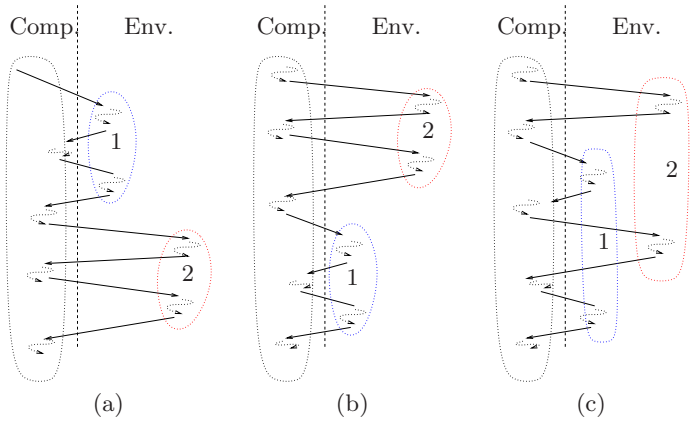


Fig. 2. Order of interaction

ture is dynamic, since communication can merge previously separate observer

cliques. After merging, the now joint clique, indicated by the big "bubble" containing 1 and 2 in Figure 3, can coordinate and thus observe the order of further interaction, but the order of past interaction cannot be reconstructed. I.e., in Figure 3, the three components, i.e., the components on the left-hand side of Figure 3(a) – 3(c) respectively, are observably equivalent.

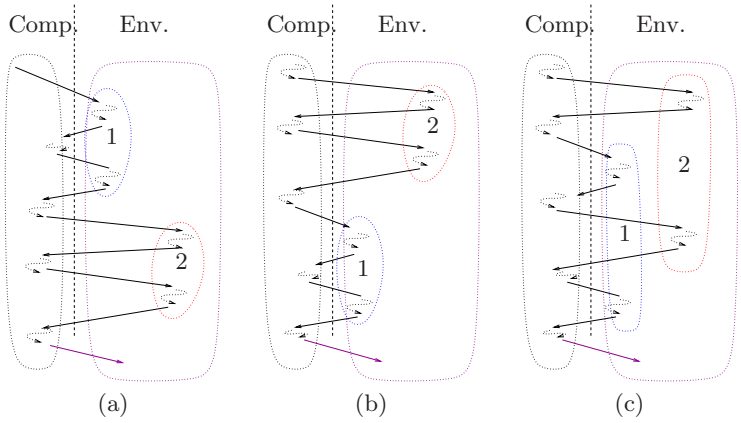


Fig. 3. Order of interaction and merging

2.3 Classes as generators of objects, replay, and determinism

Classes are generators for objects, and two instances of a class are *"identical up-to their identity"* i.e., they have the same behavior up-to renaming. If the trace of a component contains a certain behavior of an object (or more generally of a clique of objects), then it is unavoidable that the component shows a trace where the equivalent behavior is realized by a second instance of the object (or object clique): each behavior can be "replayed" on a fresh instance. With the possibility of cross-border instantiation, the component can create more than one equivalent instances of its observer, which perform equivalently.

Consider Figures 4(a) and 4(b). The second one resembles Figure 3(a) before the merge. This time, however, we assume, that the interaction s' with the first clique is a prefix of the longer $s\ t$ up-to renaming. If $s\ t$ is a possible behavior of the system, then clearly also scenario 4(b). One can use the argument also in the reverse direction: if 4(b) is possible, then so is 4(a); in other words, both behaviors are equivalent.

If afterwards the observers are merged (cf. Figure 4(c)), this scenario clearly differs from the one where the interaction s' with the formerly separate clique is missing. Unlike in the situation of Figure 3, where the order of the previously separate cliques could not be enforced in retrospect, the merging here allows to compare the different identities (but of course still not the order).

The possibility to create more than one instance from a class has a further impact when dealing with deterministic programs in the single-threaded setting. If a class is instantiated twice, its instances must behave "the same" up-to renaming, i.e., when confronted with the same input, show the same reaction. For instance, the shorter trace s' of Figure 4(b) is not only possible, given $s\ t$, but the clique on the left of 4(b) can do *nothing else* than what does the one on the right, when stimulated by the same input from the component. The scenario used environment cliques for illustration, but the same arguments apply to component cliques, as well.

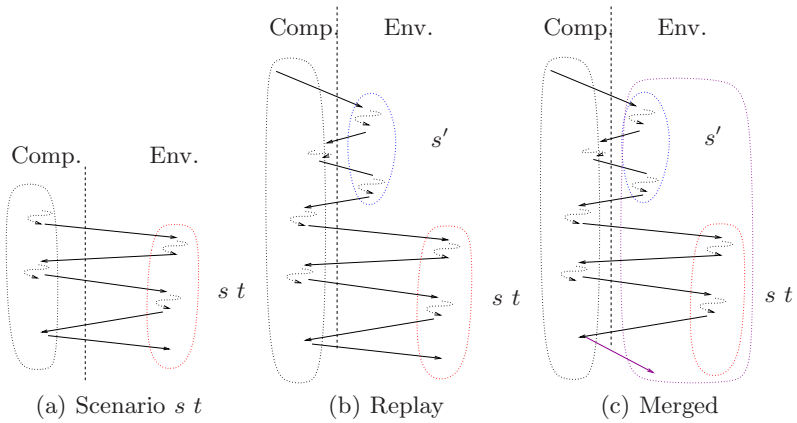
3 A single-threaded calculus with classes

Concentrating on the semantical issues, we only sketch the syntax and ignore typing issues as they are rather standard and similar to [2]. Indeed, the calculus is a restriction to single threaded programs of the one used in [2], which in turn is an extension of the concurrent object calculus from [6,10] namely by adding classes.

A program is given by a collection of classes where a class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields.[5] An object $o[c, F]$ stores the current value of the fields or instance variables and keeps a reference to the class it instantiates. A method $\varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. Besides named objects and classes, the dynamic configuration of a program contains one single thread $\natural\langle t\rangle$ as active entity.[6] A

[5] For names, we will generally use o and its syntactic variants as names for objects, c for classes, and n when being unspecific, for instance in Table 1.

[6] The $\natural$-symbol is only meant to distinguish the syntactic entity t from a running thread $\natural\langle t\rangle$.



(a) Scenario $s\ t$ (b) Replay (c) Merged

Fig. 4. Replay and merging

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

Table 1. Abstract syntax

thread basically is either a value or a sequence of expressions, notably method calls (written $v.l(\vec{v})$) and the creation of new objects $new\, c$ where c is a class name. Furthermore we will use f specifically for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update as $x.f := v$.

The *operational semantics* is given in two levels: *internal* steps whose effect is confined within a component, and those with external effect. The *external* behavior of a component is given in terms of labeled transitions describing the communication at the interface of an *open* program. For the completeness of the semantics, it is crucial ultimately to consider only communication traces realizable by an actual program context which, together with the component, yields a well-typed closed program.

Being concerned with the dynamic connectivity among objects, we omit in this paper most of the typing aspects, e.g., that transmitted values need to adhere to the static typing assumptions, that only publicly known objects can be called from the outside, and the like, since this part is rather standard and also quite similar to the one in [10].

3.1 Operational semantics

We start with component internal steps in the following section; Section 3.1.2 contains the small-step semantics describing the component-environment interaction.

3.1.1 Internal steps The internal steps are given in Table 2, where we distinguish between confluent steps, written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$.[7]

[7] In the single-threaded setting, the distinction is not too important, since at any time at most one reduction step is enabled. It may nevertheless enhance understanding to conceptually distinguish between side-effect free steps and those that may lead to race conditions when executed in the presence of other threads.

$$\natural\langle let\, x{:}T = v\, in\, t\rangle \rightsquigarrow \natural\langle t[v/x]\rangle \qquad \textsc{Red}$$

$$\natural\langle let\, x_2{:}T_2 = (let\, x_1{:}T_1 = e_1\, in\, e)\, in\, t\rangle \rightsquigarrow \natural\langle let\, x_1{:}T_1 = e_1\, in\ (let\, x_2{:}T_2 = e\, in\, t)\rangle \qquad \textsc{Let}$$

$$\natural\langle let\, x{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, in\, t\rangle \rightsquigarrow \natural\langle let\, x{:}T = e_1\, in\, t\rangle \qquad \textsc{Cond}_1$$

$$\natural\langle let\, x{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, in\, t\rangle \rightsquigarrow \natural\langle let\, x{:}T = e_2\, in\, t\rangle \qquad \textsc{Cond}_2$$

$$\natural\langle let\, x{:}T = stop\, in\, t\rangle \rightsquigarrow \natural\langle stop\rangle \qquad \textsc{Stop}$$

$$c[\![F, M]\!] \parallel \natural\langle let\, x{:}c = new\, c\, in\, t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F] \parallel \natural\langle let\, x{:}c = o\, in\, t\rangle) \qquad \textsc{NewO}_i$$

$$c[\![F, M]\!] \parallel o[c, F'] \parallel \natural\langle let\, x{:}T = o.l(\vec{v})\, in\, t\rangle \xrightarrow{\tau}$$
$$c[\![F, M]\!] \parallel o[c, F'] \parallel \natural\langle let\, x{:}T = M.l(o)(\vec{v})\, in\, t\rangle \qquad \textsc{Call}_i$$

$$o[c, F] \parallel \natural\langle let\, x{:}T = o.f := v\, in\, t\rangle \xrightarrow{\tau} o[c, F.f := v] \parallel \natural\langle let\, x{:}T = o\, in\, t\rangle \qquad \textsc{FUpdate}$$

Table 2. Internal steps

$$\mathbf{0} \parallel C \equiv C$$
$$C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2)$$
$$\nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Table 3. Structural congruence

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 3 where in the fourth axiom, n does not occur free in C_1.

3.1.2 External steps While the component-internal steps are fairly standard and straightforward, the external semantics is more complex. With the goal of full-abstraction in mind it is necessary to ultimately characterize the interaction between component and environment which is realizable by *some* program (cf. the legal traces from Section 5.2). To ease the full abstraction argument, we formulate the semantics under the assumption that the component together with the (absent or abstracted) environment gives a well-formed program adhering to the syntactical and the context-sensitive restrictions of the language at hand.

So the interface behavior is phrased in an assumption-commitment framework and based on three orthogonal abstractions:

– a static abstraction, i.e., the type system (largely omitted here);

– an abstraction of the stacks of recursive method invocations, representing the recursive and reentrant nature of method calls in a multi-threaded setting;
– finally an abstraction of the *heap topology,* approximating potential connectivity of objects and threads. The heap topology is dynamic in that new objects may be created and tree structured in that previously separate groups of objects may merge.

Connectivity contexts and cliques As discussed, in the presence of internal and external classes and cross-border instantiation, the semantics must contain a representation of the object connectivity. The external semantics is formalized as labeled transitions between judgments of the form

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

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

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

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$, where $\times$ denotes the pairing and $+$ the disjoint combination of Δ and Θ. We write $o_1 \hookrightarrow o_2$ ("o_1 may know o_2") for pairs from these relations. Without full information about the complete system, the component must make worst-case assumptions concerning the proliferation of knowledge, which are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of objects *from* Δ. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

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

In the definition, $\downarrow_\Delta$ stands for the projection of the relation onto the restricted domain Δ. Note that we close $\hookrightarrow$ only wrt. environment objects, but not wrt. objects at the *interface,* i.e., the part of $\hookrightarrow \subseteq \ \Delta \times \Theta$. We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\hookrightarrow$ for that union. As judgment, we use $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$, resp. $\Delta; E_\Delta \vdash o_1 \leftrightharpoons\!\hookrightarrow o_2 : \Theta$. For Θ, E_Θ, and Δ, the definitions are applied dually.

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

External steps The external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments in Table 5. Besides internal steps a component exchanges information with the environment via *calls* and *returns* (cf. Table 4).

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

Table 4. Labels

To formulate the external communication, we need to augment the syntax by two additional expressions o_1 *blocks for* o_2 and o_2 *returns to* $o_1\ v$. The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1. Furthermore, we augment the syntax of the method definitions accordingly, such that each method call is preceded by an annotation of the caller; i.e., instead of $\varsigma(\mathit{self}{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots)$ we write $\varsigma(\mathit{self}{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots \mathit{self}\, x.l(\vec{y})\ldots)$.

Connectivity assumptions and commitments As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ over-approximates 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 $\vec{v}$. For incoming communication (cf. rules CalliI and RetI) we require that the sender is acquainted with the transmitted arguments.

For the role of the caller identity o_1: The antecedent of the call-rules requires, that the caller o_1 is acquainted with the callee o_2 and with all of the arguments. However, the caller is *not* transmitted in the label which means that it remains anonymous to the callee.[8] To gauge, whether an incoming call is possible and to adjust the book-keeping about the connectivity appropriately. With the sole exception of the initial (external) step, the scope of at least *one* object of the calling clique must have escaped to the component, for otherwise there would be now way of the caller to address o_2 as callee. In other words, for at least one object o_1 from the clique of the actual caller (which remains anonymous), the judgment $\Delta \vdash o_1 : c$ holds prior to the call, where the judgment —the typing rules are not shown here— asserts that object o_1 carries type c according to the *environment* (and not the component) context.

While E_Δ imposes restrictions for incoming communication, the commitment context E_Θ is *updated* when receiving new information. For instance in CalliI, the commitment $\acute{E}_\Theta$ after reception marks that now the callee o_2 is acquainted

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

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ \langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(\langle call\ o_2.l(\vec{v})\rangle)\quad \vdash \Delta, \Theta : static \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow \vec{v}\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; \odot \hookrightarrow (\Delta', \Theta')\quad \Delta \vdash \odot \\ ; \acute{\Theta} \vdash o_2 : c_2\quad ; \Delta; \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]\quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T}\quad \acute{\Delta}; \acute{E}_\Delta \vdash \odot \leftrightharpoons \vec{v}, o_2 : \acute{\Theta} \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ returns\ to \odot\ x\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CalliI}_0$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ \langle call\ o_2.l(\vec{v})\rangle!\quad (\Theta', \Delta') = fn(\langle call\ o_2.l(\vec{v})\rangle \cap \Phi\quad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \\ \acute{\Delta} \vdash o_2 : c_2\quad \vdash \Delta, \Theta : static \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow \vec{v}\qquad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(C, \Theta') \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = \odot\ o_2.l(\vec{v})\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle let\ x{:}T = \odot\ blocks\ for\ o_2\ in\ t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CalloO}_0$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ \langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow \vec{v}\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \\ ; \acute{\Theta} \vdash o_2 : c_2\quad ; \Delta; \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]\quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ \acute{\Delta}; \acute{E}_\Delta \vdash o_1 \leftrightharpoons \vec{v}, o_2 : \acute{\Theta}\quad t_{blocked} = let\ x'{:}T' = o_2'\ blocks\ for\ o_1\ in\ t \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel \natural\langle t_{blocked}\rangle : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ returns\ to\ o_1\ x; t_{blocked}\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CalliI}$$

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

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ \langle call\ o_2.l(\vec{v})\rangle!\quad (\Theta', \Delta') = fn(\langle call\ o_2.l(\vec{v})\rangle \cap \Phi\quad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \\ \acute{\Delta} \vdash o_2 : c_2\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow \vec{v}\qquad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(C, \Theta') \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = o_1\ o_2.l(\vec{v})\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CalloO}$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ \langle return(v)\rangle?\qquad dom(\Delta', \Theta') \subseteq fn(v) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_1 \hookrightarrow v\qquad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow (\Delta', \Theta') \\ ; \Delta \vdash o_2 : c_2\quad ; \Delta; \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]\quad ; \acute{\Delta}, \acute{\Theta} \vdash v : T\quad \acute{\Delta}; \acute{E}_\Delta \vdash o_2 \leftrightharpoons v : \acute{\Theta} \end{array}}{\Delta; E_\Delta \vdash C \parallel \natural\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel \natural\langle t[v/x]\rangle : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{RetI}$$

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

Table 5. External steps

with the received arguments. For *outgoing* communication, the E_Δ and E_Θ play dual roles. In the respective rules, $E(\acute{C}, \Theta')$ stands for the actual connectivity of

the component after the step, which needs to be made public in the commitment context, in case new names escape to the environment.

In case of the very first interaction, either an incoming or outgoing call (cf. rules $\textsc{CallI}_0$ or $\textsc{CallO}_0$), we take $\odot$ as the source of the call, which is assumed to be resident either in the environment or the component. Furthermore, at the beginning, no objects are visible yet across the border which is asserted by $\mathit{static}(\Delta, \Theta)$. The remaining premises of the form $;\Delta \vdash n : T$ or similar deal with static typing issue, i.e., guaranteeing subject reduction. We omit the formalization of the static typing system here, as it is straightforward.

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

Rule $\textsc{NewO}_{lazy}$ describes the local instantiation of an external class. Instead of exporting the newly created name of the object plus the object itself immediately to the environment, the name is kept local until, if ever, it gets into contact with the environment. When this happens, the new instance will not only become known to the environment, but the object will also be instantiated in the environment.

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

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

4 Trace semantics and ordering on traces

Next we present the semantics for well-typed components, which takes the *traces* of the program fragment as starting point. A trace t is a sequence of external steps, i.e., given by $\Xi_1 \vdash C_1 \stackrel{s}{\Longrightarrow} \Xi_2 \vdash C_2$.

The clique structure of the environment influences what is observable and the fact that the observer falls into a number of independent groups of objects increases the "uncertainty of observation". In Section 2, we informally discussed two reasons responsible for this effect. One is that the clique of objects can only observe the order of events *projected* to its own members but not the relative order among separate cliques. Secondly, separate observers cannot cooperate to *compare identities.*

For the definition, we need to connect the labels of a trace to the clique they belong to. With the exception of the callee of a call, the communication labels do not carry information about the identity of the communication partners (cf. Table 4). Given a trace of past interaction, which adheres to a strict call-return discipline and which is strictly alternating between input and output, it contains enough information to determine the communication partners.

4.1 Balance conditions

We start with auxiliary definitions concerning the parenthetic nature of calls and returns of a legal trace (cf. Definition 1). The definition is similar to the one from [10]. It is easy to see that, starting from an initial configuration, the operational semantics from Section 3.1.2 assures strict alternation of incoming and outgoing communication and additionally that there is no return without a preceding matching call. Later, we will need this property of traces for the characterization of legal traces.

Definition 1 (Balance). *The balance of a sequence s is given by the rules of Table 6, where the dual rules for $balanced^-$ are omitted. We write $\vdash s : balanced$ if $\vdash s : balanced^+$ or $\vdash s : balanced^-$. We call a (not necessarily proper) prefix*

$$\frac{}{\vdash \epsilon : balanced^+}\ \text{B-Empty}^+$$

$$\frac{\vdash s_1 : balanced^+ \qquad \vdash s_2 : balanced^+ \qquad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : balanced^+}\ \text{B-II}$$

$$\frac{\vdash s : balanced^-}{\vdash \nu(\Phi).\langle call\ o_r.l(\vec{v})\rangle!\ s\ \nu(\Phi').\langle return(v)\rangle? : balanced^+}\ \text{B-OI}$$

Table 6. Balance

of a balanced trace weakly balanced. *We write $\vdash s : wbalanced^+$ if the trace is weakly balanced and if the last label is an incoming communication or if s is empty; dually for $\vdash s : wbalanced^-$.*

The function pop on traces is defined as follows:

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

Note that the definition of *pop*, when defined, yields a unique value. Especially, the three cases are mutually exclusive and in case 2. resp. 3, the requirement that s_2 is balanced determines $s_1 a$ uniquely.

Based on a balanced past, the following definition formalizes the notion of source and target of a communication event at the end of a trace with the help of the function *pop*.

Definition 2 (Sender and receiver). *Let $r\ a$ be a balanced trace. Sender and receiver of a after history r are defined by mutual recursion and pattern matching over the following cases:*

$$\begin{aligned} sender(\nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle!) &= \odot \\ sender(r'\ a'\ \nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle!) &= receiver(r'\ a') \\ sender(r'\ a'\ \nu(\Phi).\langle return(l(\vec{v}))\rangle!) &= receiver(pop(r'\ a')) \end{aligned}$$

$$\begin{aligned} receiver(r\ \nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle!) &= o_2 \\ receiver(r\ \nu(\Phi).\langle return(\vec{v})\rangle!) &= sender(pop(r)) \end{aligned}$$

For $a = \nu(\Phi)\langle call\ o_2.l(\vec{v})\rangle?$ resp. $a = \nu(\Phi).\langle return(\vec{v})\rangle?$, the definition is dual.

4.2 Equivalences

Now given a global trace, its projection onto one particular clique of objects as given at the end of the trace is defined straightforwardly by induction on the length of the trace. We write $[o]_{/_{E_\Delta}}$ for the equivalence class of objects according to E_Δ, i.e., the clique in connection with o, or in general just shorter $[o]$ when E_Δ is clear from the context.

Definition 3 (Projection). *Assume as trace $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$ 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 7. The definition of the projection onto a* component *clique is defined dually.*

The projection of the empty trace surely is empty (rule P-Empty). For output actions in P-Out$_1$ and P-Out$_2$ we distinguish according to the receiver, i.e., the callee in case of a call resp. the caller in case of a return. If the receiver is not involved in the communication, the label is "projected out"; dually for incoming communication. More interesting is P-Out$_2$: fresh names are not only the globally fresh ones Φ'_1, but also the locally fresh ones Φ'_2. The situation for incoming new names is *not symmetric!* It is simpler as we need not distinguish between locally and globally new names: Everything that the clique has created in isolation is globally new as well as locally new.

Besides "local freshness" we have to cater for the fact that the *order* of events cannot be determined by separate observers, i.e., we need to formalize the ideas illustrated in Section 2.2. We do this by a notion of swappability,

$$\frac{}{(\epsilon, \Phi) \downarrow_{[o]} = (\epsilon, \Phi)}\ \text{P-EMPTY}$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{receiver}(t\gamma!) \notin [o]}{(t\gamma!, \Phi) \downarrow_{[o]} = (t', \Phi')}\ \text{P-OUT}_1$$

$$\frac{\begin{array}{c}\Phi'_2 = \mathit{fn}(\nu(\Phi'_1).\gamma) \setminus \Phi' \\ (t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{receiver}(t\gamma!) \in [o]\end{array}}{(t\ \nu(\Phi'_1).\gamma!, \Phi) \downarrow_{[o]} = (t'\ \nu(\Phi'_1, \Phi'_2).\gamma!, (\Phi', \Phi'_1, \Phi'_2))}\ \text{P-OUT}_2$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{sender}(t\gamma?) \notin [o]}{(t\gamma?, \Phi) \downarrow_{[o]} = (t', \Phi')}\ \text{P-IN}_1$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{sender}(t\gamma?) \in [o]}{(t\ \nu(\Phi'').\gamma?, \Phi) \downarrow_{[o]} = (t'\ \nu(\Phi'').\gamma?, (\Phi', \Phi''))}\ \text{P-IN}_2$$

Table 7. Projection to an environment clique

where sub-sequences can be reordered when indistinguishable by the environment. This means the definition takes into account the worst-case estimations from E_Δ about the clique structure of the environment, which we indicate by the subscript[9] Δ. The dual version of the relation, written $\asymp_\Theta$, takes into account the clique structure of the component. It captures the possible reorderings of a given behavior of the component.

Whether or not the order of two actions in a trace is indistinguishable depends on clique situation of the environment at the point where the actions occur. Therefore we generalize the judgment $\Delta; E_\Delta \vdash\ o_1 \leftrightharpoons o_2 : \Theta$ from Section 3 to express acquaintance *after* executing some trace.

Definition 4 (Dynamic acquaintance). *Assume* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$. *We write* $\Delta; E_\Delta \vdash s \rhd o_1 \leftrightharpoons o_2 : \Theta; E_\Theta$, *if* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$ *and* $\acute{\Delta}; \acute{E}_\Delta \vdash o_1 \leftrightharpoons o_2 : \acute{\Theta}$. *The notation is used analogously for* $\leftrightharpoons\!\!\hookrightarrow$.

We use the definition analogously for subsequences of a trace, i.e., given $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{st_1t_2u}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \Theta; E_\Theta$, we write $\Delta; E_\Delta \vdash s \rhd t_1 \leftrightharpoons t_2 : \Theta$ if there exists a communication partner[10] o_1 of the environment mentioned in t_1 and a communication partner o_2 from t_2 acquainted according to Definition 4.

[9] The Δ is meant just as indication, that swappability is interpreted from the perspective of the environment, not as a concrete argument of the definition of $\asymp$.

[10] Sender or receiver depending on whether the action is incoming or outgoing.

$$\frac{\Xi \vdash s \rhd t_1 \not\leftrightharpoons t_2 \qquad \vdash t_1 : \mathit{balanced}}{\Xi \vdash s\ \nu(\Phi).t_1t_2u \asymp_\Delta s\nu(\Phi).t_2t_1u}\ \text{E-SwapB}_\Delta$$

$$\frac{\Xi \vdash s \rhd t_1 \not\leftrightharpoons t_2 \qquad \vdash t_1, t_2 : \mathit{wbalanced}}{\Xi \vdash s\ \nu(\Phi).t_1t_2 \asymp_\Delta s\nu(\Phi).t_2t_1}\ \text{E-SwapW}_\Delta$$

Table 8. Swapping

Definition 5 (Swapping). *The relation $\asymp_\Delta$ on traces is given as the reflexive, symmetric, and transitive closure of the rules of Table 8. The two rules silently assume that the traces involved are weakly balanced. The relation $\asymp_\Theta$ is defined dually.*

The definition of $\asymp_\Delta$ distinguishes between swapping of two neighboring subsequences in the middle of a trace (rule E-SwapB$_\Delta$) and at the end (rule E-SwapW$_\Delta$). In case of E-SwapB$_\Delta$, we require that one of the subsequences, in the rule t_1, is in itself balanced, i.e., without the preceding and trailing "contexts" s resp. t_2u. Note that t_2 is not required to be balanced, as well (but the rule can be applied symmetrically); the swapping, however, must preserve overall weak balance. That balance requirement for t_1 is needed illustrates the following consideration: Take for instance the right-hand side st_2t_1u, then moving t_2 after an unbalanced (for instance only weakly balanced) t_1 may (re-)connect returns in t_2 to unanswered calls in t_1. Similarly, returns in u may be reconnected, which means that they belong to a different environment cliques when comparing st_1t_2u and st_2t_1u. This may lead to observably different behavior. Requiring that one of the sub-sequences is balanced avoids this effect. Similar considerations imply that for swapping sub-sequences at the end, we must require *weak* balance (cf. rule E-SwapW$_\Delta$). Note that it is not sufficient that only *one* of the sub-sequences involved is weakly balanced.

Remains the formalization of the fact that different instances of the same class, or more generally different cliques identical up-to their identities, do not count as adding new behavior to the system, i.e., next we formalize the intuition from Section 2.3. The equivalence relation $\asymp$ from above is extended to consider two behaviors as equivalent if one clique is just a "replay" (up-to renaming of behavior) already witnessed in the trace. In other words: a trace can be equivalently extended by an additional action, if the behavior of the extended clique is contained as behavior of another clique already, i.e., in the form of a prefix, for which we write $\preccurlyeq$. Note that the prefix is understood up-to α-renaming.

Definition 6 (Swapping and replay). *The relation $\approx_\Delta$ on traces is given by the reflexive, transitive, and symmetric closure of the relation given in Table 9. The relation $\approx_\Theta$ is defined dually.*

We can now define the order on traces as follows.

$$\frac{receiver(s\gamma!) = o \qquad s\gamma! \downarrow_{[o]} \preccurlyeq\ s \downarrow_{[o']}}{\Delta; E_\Delta \vdash s\gamma! \asymp_\Delta s : trace\,\Theta; E_\Theta}\ \text{E-ReO}_\Delta \qquad \frac{sender(s\gamma?) = o \qquad s\gamma? \downarrow_{[o]} \preccurlyeq\ s \downarrow_{[o']}}{\Delta; E_\Delta \vdash s\gamma? \asymp_\Delta s : trace\,\Theta; E_\Theta}\ \text{E-ReI}_\Delta$$

$$\frac{\Delta; E_\Delta \vdash s \asymp_\Delta t : \Theta; E_\Theta}{\Delta; E_\Delta \vdash s \asymp_\Delta t : \Theta; E_\Theta}\ \text{E-Swap}_\Delta$$

Table 9. Swapping and replay

Definition 7. *$\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, if the following holds. If $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \overset{s}{\Longrightarrow}$, then $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta \overset{t}{\Longrightarrow}$ such that $\Delta; E_\Delta \vdash t : det_\Delta\Theta; E_\Theta$.*

5 Full abstraction

After fixing the notion of observation, we address one core problem for establishing the connection between the trace preorder and the contextual preorder, namely the characterization of legal traces, i.e., the traces which are realizable by a component together with an *arbitrary* (but well-formed, well-typed ...) context. Especially in the single-threaded setting this requires to capture deterministic traces.

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 a some notion of *observability.*

As starting point we choose, as [10], a (standard) notion of semantic equivalence or rather semantic implication —one program allows at least the observations of the other— based on a particular, simple form of contextual observation: being put into a context, the component, together with the context, reaches a defined point, which is counted as the successful observation. Being deterministic, there is no need to distinguish whether the program "may" reach the point of observation or "must" reach it. A context $\mathcal{C}[\_]$ is a program "with a hole". In our setting, the hole is filled with a program fragment consisting of a *component* C in the syntactical sense, i.e., consisting of the parallel composition of (named) classes, named objects, and named threads, and the context is the rest of the programs such that $\mathcal{C}[C]$ gives a well-typed *closed* program $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$, where closed means that it can be typed in the empty contexts, i.e., $\vdash C' : ()$.

To report success, we assume an external class with a particular success reporting method. So assume a class c_b of type $[\![succ : () \to none]\!]$, abbreviated as

barb. A component C *strongly barbs on* c_b, written $C \downarrow_{c_b}$, if $C \equiv \nu(\vec{n}{:}\vec{T}, b{:}c_b).C' \parallel \natural\langle let\ x{:}none = b.succ()\ in\ t\rangle$, i.e., the call to the success-method of an instance of c_b is enabled. Furthermore, C *barbs on* c_b, written $C \Downarrow_{c_b}$, if it can reach a point which strongly barbs on c_b, i.e., $C \Longrightarrow C' \downarrow_{c_b}$. We can now define observable preorder [7] similar as in [10]. Since the programs are deterministic, the distinction between a "may" and a "must" success disappears.

Definition 8 (Observable preorder). *Assume* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta$ *and* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$. *Then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{obs} C_2 : \Theta; E_\Theta$, *if* $(C_1 \parallel C) \Downarrow_{c_b}$ *implies* $(C_2 \parallel C) \Downarrow_{c_b}$ *for all* $\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C : \Delta; E_\Delta$.

5.2 Legal traces

As mentioned, we must characterize which traces, the "legal" ones, can occur at all, and again the crucial difference to the object-based case is to take connectivity into account to exclude impossible combinations of transmitted object names and threads. Furthermore, we need to filter out non-deterministic ones in the single-threaded setting.

The legal traces are specified by a system for judgments of the form $\Delta; E_\Delta \vdash r \rhd s : trace\,\Theta; E_\Theta$ 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. The rules are shown in Table 10. The premises of the form $;\acute{\Theta} \vdash o_2 : c_2$, $;\Delta, \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]$, and $;\acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T}$, e.g., as mentioned in rule L-CALLI, check that message exchange respects the static typing assumptions.

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

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

$$\frac{\vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \xrightarrow{\lfloor a \rfloor} o_r : ok \quad \Delta', \Theta' \vdash \lfloor a \rfloor : ok \quad a = \nu(\Phi').\ \langle return(v)\rangle? \quad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : trace\,\acute{\Theta}; \acute{E}_\Theta}{\Delta; E_\Delta \vdash r \rhd a\ s : trace\,\Theta; E_\Theta} \text{ L-RETI}$$

$$\frac{\vdash \epsilon \rhd \odot \xrightarrow{a} o_r \quad \acute{\Xi} = \Xi + \odot \xrightarrow{a} o_r \quad \acute{\Delta}, \acute{\Theta} \vdash \lfloor a \rfloor : ok \quad \acute{\Xi} \vdash \odot \xrightarrow{\lfloor a \rfloor} o_r : ok \quad \Delta_0, \Theta_0 \vdash static \quad \Delta \vdash \odot \quad a = \nu(\Delta', \Theta').\ \langle call\ o_r.l(\vec{v})\rangle? \quad \acute{\Delta}; \acute{E}_\Delta \vdash a \rhd s : \acute{\Theta}; \acute{E}_\Theta}{\Delta_0 \vdash \epsilon \rhd a\ s : trace\,\Theta_0} \text{ L-CALLI}_0$$

Table 10. Legal traces

The premise $\Delta \vdash r \rhd a : \Theta$ asserts that after r, the action a is enabled.

Definition 9 (Enabledness). *Given a method call* $\gamma = \nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle$. *Then call-*enabledness *of* γ *after the history* r *and in the contexts* Δ *and* Θ *is defined as:*

$$\Delta; E_\Delta \vdash r \rhd \gamma? : \Theta; E_\Theta \text{ if } pop\ \ r = \bot \quad and \quad \Delta \vdash \odot \text{ or } pop\ \ r = r'\gamma'! \tag{4}$$

$$\Delta; E_\Delta \vdash r \rhd \gamma! : \Theta; E_\Theta \text{ if } pop\ \ r = \bot \quad and \quad \Delta \vdash \odot \text{ or } pop\ \ r = r'\gamma'? \tag{5}$$

For return labels $\gamma = \nu(\Phi).\langle return(v)\rangle$, $\Delta; E_\Delta \vdash r \rhd \gamma! : \Theta; E_\Theta$ *abbreviates* $pop\, r = r'\nu(\Phi').\langle call\ o_2.l(\vec{v})\rangle?$, *and dually for incoming returns* $\gamma?$.

We also say, the thread is *input-call enabled* after r if $\Delta \vdash r \rhd \gamma? : \Theta$ for some incoming call label, respectively *input-return* enabled in case of an incoming return label. 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.

Being single-threaded, the language is *deterministic,* i.e., given a configuration, the next operational step is determined (up-to possible renamings). This is not only a fact about the global system behavior, but also —and more interestingly in our context— tells us that two instances of the same class, when stimulated by the same input history must react identically, up-to renaming (cf. also the discussion in Section 2.3). We thus need a characterization of deterministic traces to define when a trace is legal or not.

The issue has various aspects. That we can speak of a single trace being deterministic or not is a consequence of having classes with the possibility of cross-border instantiation and thus the possible presence of separate cliques of objects. Only then, different behaviors of "the same" object or more generally "the same" clique can show up in the trace, where the non-deterministic ones need to be filtered out to obtain an adequate characterization of the legal traces. Furthermore, the intuition "determinism means the same reaction to the same stimulus" needs some fleshing out. The past of a clique is the projection of the global trace onto the clique, which is, as usual, considered only up-to α-renaming. Furthermore, the dynamic nature of the clique structure has to be taken into account; for instance, the histories corresponding to Figure 3(a) – 3(c) are to be considered equivalent because the order of events of previously separate sub-cliques of a given clique cannot be reconstructed in retrospect.

The mentioned ideas are captured in the $\asymp$ relation, which we can use in the following definition.

Definition 10 (Deterministic trace). *Given the label* $a = \gamma!$ *and a trace* ra *with* $\Delta \vdash r \rhd a : \Theta$. *The trace* r *can be extended deterministically by* a, *written* $\Delta \vdash r \rhd a : det_\Theta\ \Theta$, *if the following holds:*

$$\begin{array}{l} \Delta; E_\Delta \vdash ra \asymp_\Theta r : \Theta; E_\Theta \quad or \\ there\ does\ not\ exist\ a\ label\ b\ with \quad \Delta; E_\Delta \vdash rb \asymp_\Theta r : \Theta; E_\Theta \end{array} \tag{6}$$

The definition for incoming communications a is dual, and especially refers to $\preccurlyeq_\Delta$ *instead of* $\preccurlyeq_\Theta$.

Note that the condition from Equation (6) does not in itself guarantee determinism for the trace; if the shorter r is deterministic, it preserves determinism when extending the trace, which is the way, the check is used in the legal trace system. We use the judgment $\Delta; E_\Delta \vdash r \rhd a : det_\Theta\ \Theta; E_\Theta$ to combine enabledness and the output determinism requirement for the next action in a single assertion. Dually we use det_Δ for input determinism for incoming communication. We write also $\Delta; E_\Delta \vdash s : det_\Delta \Theta; E_\Theta$ resp. $\Delta; E_\Delta \vdash t : det_\Theta \Theta; E_\Theta$, when the whole trace is deterministic wrt. the environment, resp. wrt. component.

5.3 Soundness and completeness

The proof that the observational order coincides with the order on traces given in Definition 7 has two directions: compared to $\sqsubseteq_{obs}$, the relation $\sqsubseteq_{trace}$ is neither too abstract (soundness) nor too concrete (completeness). For lack of space, we simply state the soundness result here.

For correspondence of the two notions is guaranteed only when assuming that the environment behaves deterministic. Therefore we refine the definition of $\sqsubseteq_{trace}$ from Definition 7, in that we explicitly require that the traces compared by $\sqsubseteq_{trace}$ are deterministic wrt. the environment; we write $\sqsubseteq^{det}_{trace}$ for that relation. The reason is that the external operational semantics of Table 5 results in deterministic behavior as far as the component is concerned —one cannot program non-deterministic behavior with the given syntax— but not for the environment. One could have checked deterministic environment behavior in the assumptions of the operational rule; the price for this more exact representation of possible behavior would have been to augment the semantics to contain the *history* of past interaction concerning the environment behavior, in a similar way as we have done when formalizing the legal traces.

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

Completeness asserts the reverse direction:

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

At the heart, completeness is a constructive argument: given a trace s, construct a component C_s that exhibits the trace s and moreover realize it *exactly*. Restricted to deterministic traces, the proof is rather similar to the one for the multi-threaded case and rests on the ability to compose a component and an environment, performing complementary traces, into one global program (plus the dual property of decomposition). Indeed, the very same construction could be used in the single-threaded setting as in the multi-threaded setting. However, the absence of concurrency allows to simplify the construction, in particular, one can leave out the code that assures mutual exclusion, when accessing objects, resp. cliques of objects.

6 Conclusion

Related work Smith [17] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* specification language. called the *complete-readiness* model, related to the readiness model of Olderog and Hoare. [18] investigates full abstraction in an object calculus with subtyping. The setting is a bit different from the one as used here as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. Recently, Jeffrey and Rathke [11] extended their work on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*. Cf. also [14]. [5] tackles the problem of full abstraction and observable component behavior and connectivity in a UML-setting. Unlike this contribution, [5] features concurrency

Future work The trace semantics together with the equivalence relation capturing the undefinednes of order of interacting with separe cliques is a "*tree*" semantics. As illustrated also by the informal examples of Section 2, the semantics more precisely can be understood as a forest of interactions, where each tree represents one current clique of objects. As shown in this paper, the cliques can be dynamically created and the branching structure is cause by merging of cliques. We are currently working on a *direct* tree representation of the semantics. The resulting semantic is is simpler as it can do without the secondary notion of equivalence relation on traces, and furthermore one can avoid an explicit representation of object connectivity as. However, e.g., the derivation system for legal traces gets more involved in that it must reflect the branching structure.

Game theory has in recent years been successfully employed for (fully abstract) semantics of open system ("game semantics"). Cf. for instance [3] for an introduction. It seems interesting to capture our set-up especially the connectivity contexts in a game semantical framework.

Acknowledgements We thank Harald Fecher and Marcel Kyas for stimulating discussions on various aspects of this topic.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Li [12], pages 38–52.
3. S. Abramsky. Algorithmic game semantics: A tutorial introduction. In Schichtenberg and Steinbruggen [16], pages 21–47.

4. M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors. *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
5. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [4]. To appear.
6. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In Nestmann and Pierce [13].
7. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
8. IEEE. *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*. Computer Society Press, July 1998.
9. IEEE. *Seventeenth Annual Symposium on Logic in Computer Science (LICS) (Copenhagen, Denmark)*. Computer Society Press, July 2002.
10. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In LICS'02 [9].
11. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In Sagiv [15], pages 423–438.
12. Z. Li, editor. *Proceedings of the First International Colloquium on Theoretical Aspects of Computing, ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
13. U. Nestmann and B. C. Pierce, editors. *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
14. J. Rathke. A fully abstract trace semantics for a core Java language (preliminary title). In Bosangue et al. [4]. To appear.
15. M. Sagiv, editor. *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
16. H. Schichtenberg and R. Steinbruggen, editors. *Proof and System Reliability, Summer School (Marktoberdorf, Germany, 2001)*, Series F: Computer and System Sciences. NATO Advanced Study Institute, Kluwer Academic Publishers, 2001.
17. G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
18. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In LICS'98 [8].