Classes, Object Connectivity, and Observability

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

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

Abstract. We sketch the observational changes, from the viewpoint of a fully-abstract semantics, when one adds classes to a setting consisting only of objects, as in traditional object calculi.

1 Introduction

Concurrent object calculi have been investigated as a mathematical basis for imperative, object-oriented languages with multithreading and heap-allocated objects. In the context of concurrent, *object-based* programs and starting from may-testing as a very simple notion of observation, Jeffrey and Rathke [5] provide a fully abstract trace semantics for the language. Their result roughly says that, given a *component* as a set of objects and threads, its 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 a 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 [5] is developed within the concurrent ν-calculus [3], an extension of the sequential ν-calculus [7] which belongs to the tradition of various object calculi [1] and also of the π-calculus [6,8]. One distinctive feature of the calculus is that it is *object-based,* which in particular means that there are no *classes* as templates for new objects. This is in contrast to the mainstream of object-oriented languages where the code is organized in classes. 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 apart from objects, which are the passive entities, and threads, which are the active entities, now *classes* come into play.

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 pure 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 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.

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.

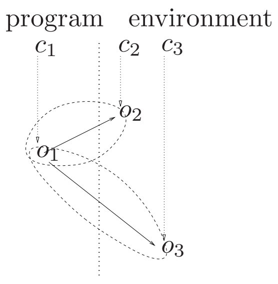

Fig. 1. Instances of external classes

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, then 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, whenever the component leaks the identity of a member of one clique to a member of another.

This extended abstract works with a class-based variant of the concurrent object calculus, sketching how to formalize the ideas mentioned above about cliques of objects, and mentions consequences concerning what is observable about a program. For want of space, we concentrate here on the intuition and refer to the technical report [2] for a deeper coverage.

2 A concurrent class calculus

Concentrating on the semantical issues of connectivity of objects, we omit the exact syntax and ignore typing issues. The calculus is a syntactic extension of the concurrent object calculus from [3,5]. The basic change is the introduction of *classes*: A program is given by a collection of classes. A class $c[\![O]\!]$ carries a name c and defines the implementation of its methods, and analogously for objects. A method $\varsigma(n{:}T).\lambda(x_1{:}T_1, \ldots, x_n{:}T_k).t$ provides the definition of the method body abstracted over the formal parameters of the method and the ς-bound "self" parameter [1]. Besides named objects and classes, the dynamic configuration of a program can contain as active entities *named threads* $n\langle t\rangle$, which, like objects, can be dynamically created. 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.

Concerning the *operational semantics* of the calculus, the basic steps are given in two levels: *internal* steps whose effect is completely confined within a configuration (which we elide), and those with external effect.

2.1 Connectivity contexts and cliques

The *external* behavior of a component is given in terms of labeled transitions describing the communication at the interface of an *open* program. As argued in the introduction, it's crucial in the presence of internal and external classes and cross-border instantiation to keep track of connectivity of objects amongst each other.

The external semantics is formalized as labeled transitions between judgments of the form $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$, 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 book-keeping "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, where $E_\Delta \subseteq \Delta \times (\Delta + \Theta)$ and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$. We will write $o_1 \hookrightarrow o_2$ ("o_1 may know o_2") for pairs from these relations. The pairs are interpreted as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs from Δ, i.e., given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

$$\leftrightharpoons \;\triangleq\; (\hookrightarrow\downarrow_\Delta \cup \hookleftarrow\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ .$$

We will also need the union of $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, for which we will also write $\leftrightharpoons\hookrightarrow$.

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 *clique* a set of object names from Δ such that for all objects o_1 and o_2 from that set, $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$ (or dually from Θ), and we speak of *the* clique of an object when we mean the whole equivalence class.

2.2 Keeping track of connectivity

The component exchanges information with the environment via *calls* and *returns* and the external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments. We show as an example the reception of a method call (fielding a return works similar and for outgoing communication, the situation is dual):

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

The contexts play dual roles: E_Θ overapproximates the actual connectivity of the component, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. For incoming calls we require that the sender be acquainted with the transmitted arguments. In case of a call, the caller o_1 must additionally be acquainted with the callee o_2 and, furthermore, the calling thread must originate from a clique of objects connected with the one to which the thread had left the component the last time: $\Delta; E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta$.[5] While E_Δ imposes restrictions for incoming communication, the commitment

[5] Actually, the caller itself remains anonymous, but the semantics must record the calling clique.

context E_Θ is *updated* when receiving new information. For instance in CallI, the commitment $\acute{E}_\Theta$ after reception marks that now the callee o_2 is acquainted with the received arguments and furthermore that the thread n is visiting (for the time being) the callee o_2.

2.3 Scoping and lazy instantiation

Since new objects and threads can be created, the semantics has to take care of *dynamic scoping* and the exchange of bound names. Apart from the connectivity, this is done in a standard way, as in the object-calculus or the π-calculus. We only mention a few particularities relevant in our context:

– Object creation across component boundaries is caused by the fact that the component instantiates an environment class (or vice versa). This behavior is absent in an object-based setting.
– Cross-border instantiation itself in *unobservable,* which is due to the fact that the calculus does not feature constructor methods and that the heap space is unbounded. As a consequence, there are no transitions labeled for object creation; rather objects are created only when they are first referenced ("lazy instantiation").
– For newly learnt objects, also connectivity information must be exchanged, i.e., the usual mechanism of scope extrusion must be enhanced by handling connectivity information, as well.

2.4 Observability

Now we return to the question posed in the beginning: What is *observable* in that class-based framework? As notion of observation, we take *may-testing* [4] which roughly is defined as follows: Put a component or program fragment into a context, let it run, and observe whether it is possible ("may") that the program reaches a success-reporting state. From this conventional, observational starting point, the question arises: what is the corresponding denotational semantics? As in the object-based setting, this gives rise to trace semantics, i.e., the behavior of a component is characterized by sets of all its interactions with the environment. Again, we only mention the salient differences to the pure object-based setting:

Connectivity: As outlined, an important change of the semantics is the need to represent connectivity, i.e., groups of objects that may know each other. The groups of objects are dynamic, in that new cliques can appear by cross-border instantiation, and they might merge. The semantics must contain only traces consistent with the connectivity structure.

Independent observers: The observing environment may be split into separate cliques of objects. Unable to coordinate their observations, the total order of interaction between different observer cliques cannot be fully determined. If one ignores deadlock and divergence, the semantics falls apart in separate behaviors per clique.

Instantiation of classes: The semantics must be aware of the fact, that two instances of the same class are initially equivalent (up-to their identity). This means, they may behave equivalently, until they are subject to communication with the environment that makes a distinction possible.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Aug. 2003.
3. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
4. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
5. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.
6. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
7. A. M. Pitts and D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In A. M. Borzyszkowski and S. Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Sept. 1993.
8. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.