Inheritance and Observability

Erika Ábrahám, Thi Mai Thuong Tran, and Martin Steffen

RWTH Aachen, Germany and University of Oslo, Norway

An *open* system is a part of a larger system, which interacts with its environment, and best considered as a black box where the internals are hidden. Such a separation of internal behavior from externally relevant interface behavior is crucial for compositionality. The most popular programming paradigm nowadays is object orientation, which in particular supports interfaces and encapsulation of objects. Another crucial feature in mainstream object orientation is *inheritance,* which allows code reuse and is intended to support incremental program development by gradually extending and specializing an existing class hierarchy.

Openness of a system in the presence of inheritance and late binding is problematic. One symptom of that is known in software engineering as the fragile base class problem. A base class in an inheritance hierarchy is a (common) super-class, and fragile means that replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class, i.e., change the behavior of the "environment" of the base class.

A rigorous method to keep track of interactional behaviours of an open program is the key to formal verification of open programs as well as a formal foundation for black-box testing. If done properly, it ultimately allows compositional reasoning, i.e., to infer properties of a composed system from the interface properties of its sub-constituents without referring to further internal representation details. A representation-independent, abstract account of the behavior is also necessary for compositional optimization of components: only when showing the same external behavior one program can replace another without changing the interaction with any client code.

An object-oriented, concurrent calculus The calculus presented in this work is a concurrent variant of an imperative, object-calculus. Its concurrency model is based on the notion of "active objects" and asynchronous method calls. Its syntax is sketched in Table 1 focusing on important features, such as: classes with fields and methods, objects as instances of classes, and concurrency based on the active objects model of concurrency. Expressions e basically consists of a sequential composition (here represented by the let-construct) of basic expressions, including conditionals, object creation, read and write of object fields, and method calls.

Being standard, the syntax should be largely clear, a few points are worth highlighting though: the concurrency model of the calculus based on *active objects,* communicating via *asynchronous* method calls (written $o@l(\boldsymbol{v})$) and the result is given back by the caller querying a future reference. Objects act as monitors with binary locks. The notion of single inheritance based on classes, however, is orthogonal to the choice of the concurrency model.

C	$::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!] \mid \underline{n[O,\mathtt{lock}]} \mid \underline{n\langle t\rangle}$	component
O	$::= n, M, F$	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	$::= v \mid \bot_{n'}$	field
t	$::= v \mid \mathtt{stop} \mid \mathtt{let}\ x{:}T = e\ \mathtt{in}\ t$	thread
e	$::= t \mid \mathtt{if}\ v = v\ \mathtt{then}\ e\ \mathtt{else}\ e \mid \mathtt{if}\ \mathit{undef}(v.l())\ \mathtt{then}\ e\ \mathtt{else}\ e$	expr.
	$\mid\ n@l(\boldsymbol{v}) \mid v.l() \mid v.l() := v$	
	$\mid\ \mathtt{new}\ n \mid \mathsf{claim}@(n,n) \mid \underline{\mathsf{get}@n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
v	$::= x \mid n \mid ()$	values
$\mathtt{lock}$	$::= \bot \mid \top$	lock status

Table 1. Syntax of an oo core calculus

Typed operational semantics for interface behavior In this setting, the component behavior consists of message traces, i.e., sequences of component-environment interactions. Writing $C \stackrel{t}{\Longrightarrow} \acute{C}$, the t denotes the *trace* of interface actions by which C evolves into $\acute{C}$, potentially executing internal steps, as well, not recorded in t. An open program C, however, does not act in isolation, but interacts with *some* environment. I.e., we are interested in traces t where *there exists an environment* E such that $C \parallel E \xRightarrow[\bar{t}]{t} \acute{C} \parallel \acute{E}$ by which we mean: component C produces the trace t and E produces the dual trace $\bar{t}$, both together "canceling out" to internal steps. In other words, our goal is to formulate the external or open semantics with the environment *existentially abstracted away.* With infinitely many possible environments E, the challenge is to capture what is common to *all* those environments. This will be done in form of *assumptions* about the environment. This means, the operational semantics specifies the behavior of C under certain assumptions $\varXi_E$ about the environment. Following standard notation from logics, we do not write $\varXi_E \parallel C$, but rather $\varXi_E \vdash C$, such that the reductions will look like

$$\varXi_E \vdash C \stackrel{t}{\Longrightarrow} \acute{\varXi}_E \vdash \acute{C}\ . \tag{1}$$

Such a characterization of the abstract interface behavior is relevant for the following reasons. Firstly: the set of traces according to equation (1) is more restricted than the one obtained when ignoring the environments altogether. This means, when *reasoning* about the behavior of C based on the traces, e.g., for the purpose of verification, the more precise knowledge of the possible traces allows to carry out stronger arguments about C. Secondly, an application for a trace description is black-box testing, in that one describes the behavior of a component in terms of the interface traces and then synthesize appropriate test drivers from it. Obviously it makes no sense to specify interface behavior which is not possible, at all, since in this case one could not generate a corresponding tester. Finally, and not as the least gain, the formulation gives *insight* into the inherent semantical nature of the language, as the assumptions Ξ and the semantics captures the existentially abstracted environment behavior.

Main results

– A formal, open semantics for a statically typed, concurrent object-oriented calculus with dynamic object creation, mutable heap, and single *inheritance.*
– The main insight of our work is that the cross-border inheritance complicates the observable behavior considerably. Namely, in an open setting, environment and component classes can inherit from each other. Therefore an object may contain fields defined by the components and by the environment. Due to privacy restrictions, these fields can only be manipulated by the corresponding methods of environment resp. component parts. To describe the possible interface behavior, where all possible environments are existentially abstracted away and represented by an assumption context, the potential connectivity of the enviroment part of the heap is important. In order to capture that, our open semantics must be able to tell when a communication between two objects is possible, i.e, when they are potentially in connnection.
– The interface behavior is characterized in the form of a typed operational semantics of an open system, consisting of a set of classes.
– The semantics is formalized in the form of commitments of the component and in particular *assumptions* about the environment. The fact that the components are open wrt. inheritance, i.e., a component can inherit from the environment and vice versa, has as a consequence that the assumptions and commitments need contain an abstraction of the heap topology, keeping track of which object may be in connection with other objects.
– Finally, we show the soundness of the abstractions based on trace semantics.

More details can be found in [1]

References

1. E. Ábrahám, T. Mai Thuong Tran, and M. Steffen. Observable interface behavior and inheritance. Technical Report 409, University of Oslo, Dept. of Informatics, Apr. 2011. `www.ifi.uio.no/~msteffen/publications.html#techreports`.