

State Based Ownership, Reentrance, and Encapsulation

Anindya Banerjee^{1,*} and David A. Naumann^{2,**}

¹ Kansas State University, Manhattan KS 66506 USA
ab@cis.ksu.edu

² Stevens Institute of Technology, Hoboken NJ 07030 USA
naumann@cs.stevens.edu

Abstract. A properly encapsulated data representation can be revised for refactoring or other purposes without affecting the correctness of client programs and extensions of a class. But encapsulation is difficult to achieve in object-oriented programs owing to heap based structures and reentrant callbacks. This paper shows that it is achieved by a discipline using assertions and auxiliary fields to manage invariants and transferrable ownership. The main result is representation independence: a rule for modular proof of equivalence of class implementations.

1 Introduction

You are responsible for a library consisting of many Java classes. While fixing a bug or refactoring some classes, you revise the implementation of a certain class in a way that is intended not to change its observable behavior, e.g., an internal data structure is changed for reasons of performance. You are in no position to check, or even be aware of, the many applications that use the class via its instances or by subclassing it. In principle, the class could have a full functional specification. It would then suffice to prove that the new version meets the specification. In practice, full specifications are rare. Nor is there a well established logic and method for modular reasoning about the code of a class in terms of the specifications of the classes it uses, without regard to their implementations or the users of the class in question [20] (though progress has been made). One problem is that encapsulation, crucial for modular reasoning about invariants, is difficult to achieve in programs that involve shared mutable objects and reentrant callbacks which violate simple layering of abstractions. Yet complicated heap structure and calling patterns are used, in well designed object-oriented programs, precisely for orderly composition of abstractions in terms of other abstractions.

There is an alternative to verification with respect to a specification. One can attempt to prove that the revised version is behaviorally equivalent to the original. Of course their behavior is not identical, but at the level of abstraction of source code (e.g., modulo specific memory addresses), it may be possible to show equivalence of behavior. If any specifications are available they can be taken into account using assert statements.

There is a standard technique for proving equivalence [18, 24]: Define a *coupling relation* to connect the states of the two versions and prove that it has the *simulation*

* Supported in part by NSF grants CCR-0209205, ITR-0326577, and CCR-0296182.

** Supported in part by NSF grants CCR-0208984, CCF-0429894, and by Microsoft Research.

property, i.e., it holds initially and is preserved by parallel execution of the two versions of each method. In most cases, one would want to define a *local coupling* relation for a single pair of instances of the class, as methods act primarily on a target object (self) and the *island* of its representation objects; an *induced coupling* for complete states is then obtained by a general construction. A language with good encapsulation should enjoy an *abstraction or representation independence* theorem that says a simulation for the revised class induces a simulation for any program built using the class. Suitable couplings are the identity except inside the abstraction boundary and an *identity extension lemma* says simulation implies behavioral equivalence of two programs that differ only by revision of a class. Again, such reasoning can be invalidated by heap sharing, which violates encapsulation of data, and by callbacks, which violate hierarchical control structure.

There is a close connection between the equivalence problem and verification: verification of object oriented code involves object invariants that constrain the internal state of an instance. Encapsulation involves defining the invariant in a way that protects it from outside interference so it holds globally provided it is preserved by the methods of the class of interest. Simulations are like invariants over two copies of the state space, and again modular reasoning requires that the coupling for a class be independent from outside interference. *The main contribution of this paper is a representation independence theorem using a state-based discipline for heap encapsulation and control of callbacks.*

Extant theories of data abstraction assume, in one way or another, a hierarchy of abstractions such that control does not reenter an encapsulation boundary while already executing inside it. In many programming languages it is impossible to write code that fails to satisfy the assumption. But it is commonplace in object oriented programs for a method m acting on some object o to invoke a method on some other object which in turn leads to invocation of some method on o —possibly m itself—while the initial invocation of m is in progress. This makes it difficult to reason about when an object's invariant holds [20, 25]; we give an example later.

There is an analogous problem for reasoning with simulations. In previous work [2] we formulated an abstraction theorem that deals with sharing and is sound for programs with reentrant callbacks, but it is not easy to apply in cases where reentrant callbacks are possible. The theorem allows the programmer to assume that all methods preserve the coupling relation when proving simulation, i.e., when reasoning about parallel execution of two versions of a method of the class of interest. This assumption is like verifying a procedure implementation under the assumption that called procedures are correct. But the assumption that called methods preserve the coupling is of no use if the call is made in an uncoupled intermediate state. For the examples in [2], we resort to ad hoc reasoning for examples involving callbacks.

In a recent advance, [6, 21] reentrancy is managed using an explicit auxiliary (or *ghost*) field *inv* to designate states in which an object invariant is to hold. Encapsulation is achieved using a notion of ownership represented by an auxiliary mutable field *own*. This is more flexible than type-based static analyses because the ownership invariant need only hold in certain flagged states. Heap encapsulation is achieved not by disallowing boundary-crossing pointers but by limiting, in a state-dependent way, their use.