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.

A.P. Black (Ed.): ECOOP 2005, LNCS 3586, pp. 387–411, 2005.

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.

Reasoning hinges on a global *program invariant* that holds in all states, using *inv* fields to track which object invariants are temporarily not in force because control is within their encapsulation boundary. When *inv* holds, the object is said to be *packed*; a field may only be updated when the object is unpacked.

In this paper we adapt the *inv/own* discipline [6, 21][1] to proving class equivalence by simulation. The *inv* fields make it possible for an induced coupling relation to hold at some pairs of intermediate states during parallel execution of two alternative implementations. This means that the relation-preservation hypothesis of the abstraction theorem can be used at intermediate states even when the local coupling is not in force. So per-method modular reasoning is fully achieved. In large part the discipline is unchanged, as one would hope in keeping with the idea that a coupling is just an invariant over two parallel states. But we have to adapt some features in ways that make sense in terms of informal considerations of information hiding. The discipline imposes no control on field reads, only writes, but for representation independence we need to control reads as well. The discipline also allows ownership transfer quite freely, though it is not trivial to design code that correctly performs transfers. For representation independence, the transfer of previously-encapsulated data to clients (an unusual form of controlled "rep exposure" [16]) is allowed but must occur only in the code of the encapsulating class; even then, it poses a difficult technical challenge. The significance of our adaptations is discussed in Section 7.

A key insight is that, although transferring ownership and packing/unpacking involve only ghost fields that cannot affect program execution, it is useful to consider them to be observable. It is difficult to reason about two versions of a class, in a modular way, if they differ in the way objects cross the encapsulation boundary or in which methods assume the invariant is in force. The requisite similarity can be expressed using assert statements so we can develop a theory based on this insight without the need to require that the class under revision has any specifications.

Contributions. The main contributions are (a) formulation of a notion of instance-based coupling analogous to invariants in the *inv/own* discipline; (b) proof of a representation independence theorem for a language with inheritance and dynamic dispatch, recursive methods and callbacks; mutable objects, type casts, and recursive types; and (c) results on identity extension and use of the theorem to prove program equivalence. Together these constitute a rule by which the reasoner considers just the methods of the revised class and concludes that the two versions yield equivalent behavior for any program context.

The theorem allows ownership transfers that cross encapsulation boundaries: from client to abstraction [16], between abstractions, and even from abstraction to client [29, 4]. The theorem supports the most important form of modularity: reasoning about one method implementation (or rather, one corresponding pair) at a time —on the assumption that all methods preserve the coupling (even the one in question, modulo termination). Our theorem also supports local reasoning in the sense that a single instance

[1] Called the "Boogie methodology" in the context of the Spec# project [7] at Microsoft Research, which implements the discipline as part of a comprehensive verification system inspired by the ESC projects.

(or pair of instances) is considered, together with the island comprised of its currently encapsulated representation objects.

The discipline can be used in any verification system that supports ghost variables and assertions. So our formalism treats predicates in assertions semantically, avoiding ties to any particular logic or specification formalism.

Related Work Besides the inv/own Discipline. Representation independence is needed not only for modular proof of equivalence of class implementations but also for modular reasoning about improvements (called data refinement). Such reasoning is needed for correctness preserving refactoring. The refactoring rules of Borba et al. [10] were validated using the data refinement theory of Cavalcanti and Naumann [13] which does not model sharing/aliasing. We plan to use the present result to overcome that limitation. Representation independence has also been used to justify treating a method as pure if none of its side effects are visible outside an encapsulation boundary [8, 26].

Representation independence is proved in [2] for a language with shared mutable objects on the basis of ownership confinement imposed using restrictions expressed in terms of ordinary types; but these restrictions disallow ownership transfer. The results are extended to encompass ownership transfer in [4] but at the cost of substantial technical complications and the need for reachability analysis at transfer points, which are designated by explicit annotations. Like the present paper, our previous results are based on a semantics in which the semantics of primitive commands is given in straightforward operational terms. It is a denotational semantics in that a command denotes a state transformer function, defined by induction on program structure. To handle recursion, method calls are interpreted relative to a *method environment* that gives the semantics of all methods. This is constructed as the limit of approximations, each exact up to a certain maximum calling depth. This model directly matches the recursion rule of Hoare logic, of which the abstraction theorem is in some sense a generalization.

For simple imperative code and single-instance modules, O'Hearn *et al.* [29, 23] have proved strong rules for local reasoning about object invariants and simulations using separation logic which, being state based, admits a notion of ownership transfer.

Confinement disciplines based on static analysis have been given with the objective of encapsulation for modular reasoning, though mostly without formal results on modular reasoning [14, 11]. Work using types makes confinement a *program invariant*, i.e., a property required to hold in every reachable state. This makes it difficult to transfer ownership, due to temporary sharing at intermediate states. Most disciplines preclude transfer (e.g., [15, 11]); where it is allowed, it is achieved using nonstandard constructs such as destructive reads and restrictive linearity constraints (e.g., [12, 30]).

Outline. Sect. 2 sketches the *inv/own* discipline. It also sketches an example of the use of simulation to prove equivalence of two versions of a class involving reentrant callbacks, highlighting the problems and the connection between our solution and the *inv/own* discipline. Sect. 3 formalizes the language for which our result is given and Sect. 4 formalizes the discipline in our semantics. Sect. 5 gives the main definitions—proper annotation, coupling, simulation—and the abstraction theorem. Sect. 6 connects simulation with program equivalence. Sect. 7 discusses future work and assesses our adaptation of the discipline. For lack of space, all proofs are omitted and can be found in the companion technical report, which also treats generics [5].

2 Background and Overview

2.1 The *inv*/*own* Discipline

To illustrate the challenge of reentrant callbacks as well as the state based ownership discipline, we consider a class Queue that maintains a queue of tasks. Each task has an associated limit on the number of times it can be run. Method Queue.runAll runs each task that has not exceeded its limit. For simplicity we refrain from using interfaces; class Task in Fig. 1 serves as the interface for tasks. Class Qnode in the same Figure is used by Queue which maintains a singly linked list of nodes that reference tasks. Field count tracks the number of times the task has been run. For brevity we omit initialization and constructors throughout the examples.

Fig. 2 gives class Queue. One intended invariant of Queue is that no task has been run more times than its limit. This is expressed, in a decentralized way, by the invariant declared in Qnode. Some notation: we write $\mathscr{I}^{Qnode}(o)$ for the predicate o.tsk≠null and o.count≤o.limit.

Another intended invariant of Queue is that runs is the sum of the count fields of the nodes reached from tsks. This is the declared $\mathscr{I}^{Queue}$ of Fig. 2. (The reader may think of other useful invariants, e.g., that the list is null-terminated.) Note that at intermediate points in the body of Queue.runAll, $\mathscr{I}^{Queue}$ does not hold because runs is only updated after the loop. In particular, $\mathscr{I}^{Queue}$ does not hold at the point where p.run() is invoked.

```
class Task {   void run(){ }   }
class Qnode {
  Task tsk;   Qnode nxt;   int count, limit;
  invariant tsk ≠ null ∧ 0≤count≤limit;
  ... // constructor elided (in subsequent figures these ellipses are elided too)
  void run() { tsk.run(); count := count+1; }
  void setTsk(Task t, int lim) {
    tsk := t; limit := lim; count := 0; pack self as Qnode; }   }
```

Fig. 1. Classes Task and Qnode. The pack statement is discussed later

```
class Queue {
  Qnode tsks;   int runs := 0;
  invariant runs = (Σp ∈ tsks.nxt* | p.count);
  int getRuns() { result := runs; }
  void runAll() {
    Qnode p := tsks; int i := 0;
    while p ≠ null do {
      if p.getCount() < p.getLimit() then p.run(); i := i+1; fi; p := p.getNxt(); }
    runs := runs+i; }
  void add(Task t, int lim){
    Qnode n := new Qnode; n.setTsk(t,lim); n.setNxt(tsks); tsks := n; } }
```

Fig. 2. Class Queue

For an example reentrant callback, consider tasks of the following type.

class RTask extends Task { Queue q; **void** run(){q.runAll(); } ...}

Consider a state in which o points to an instance of Queue and the first node in the list, o.tsks, has count=0 and limit=1. Moreover, suppose field q of the first node's task has value o. Invocation of o.runAll diverges: before count is incremented to reflect the first invocation, the task makes a *reentrant call* on o.runAll —in a state where $\mathscr{I}^{Queue}$ does not hold. In fact runAll again invokes run on the first task and the program fails due to unterminating recursion.

As another example, suppose RTask.run is instead **void** run(){q.getRuns();} . This seems harmless, in that getRuns neither depends on $\mathscr{I}^{Queue}$ nor invokes any methods —it is even useful, returning a lower bound on the actual sum of runs. It typifies methods like state readers in the observer pattern, that are intended to be invoked as reentrant callbacks.

The examples illustrate that it is sometimes but not always desirable to allow a reentrant callback when an object's invariant is violated temporarily by an "outer" invocation. The ubiquity of method calls makes it impractical to require an object's invariant to be reestablished before making *any* call —e.g., the point between n.setTsk and n.setNxt of method add in Fig. 2 — although this is sound and has been proposed in the literature on object oriented verification [17, 22].

A better solution is to prevent just the undesirable reentrant calls. One could make the invariant an explicit precondition, e.g., for runAll but not getRuns. This puts responsibility on the caller, e.g., RTask.run cannot establish the precondition and is thus prevented from invoking runAll. But an object invariant like $\mathscr{I}^{Queue}$ involves encapsulated state not suitable to be visible in a public specification.

The solution of the Boogie methodology [6, 21] is to introduce a public ghost field, *inv*, that explicitly represents whether the invariant is in force. In the lingo, $o.inv$ says object o is *packed*. Special statements **pack** and **unpack** set and unset *inv*.

A given object is an instance not only of its class but of all its superclasses, each of which may have invariants. The methodology takes this into account as follows. Instead of *inv* being a boolean, as in the simplified explanation above, it ranges over class names C such that C is a superclass of the object's allocated type. That is, it is an invariant (enforced by typing rules) that $o.inv \geq type(o)$ where $type(o)$ is the dynamic type of o. The discipline requires certain assertions preceding pack and unpack statements as well as field updates, to ensure that the following is a *program invariant* (i.e., it holds in all reachable states).

$$o.inv \leq C \Rightarrow \mathscr{I}^C(o) \tag{1}$$

for all C and all allocated objects o. That is, if o is packed at least to class C then the invariant $\mathscr{I}^C$ for C holds. Perhaps the most important stipulated assertion is that $\mathscr{I}^C(o)$ is required as precondition for packing o to level C.

Fig. 3 shows how the discipline is used for class Queue. Assertions impose preconditions on runAll and add which require that the target object is packed to Queue. In runAll, the **unpack** statement sets *inv* to the superclass of Queue, putting the task in a position where it cannot establish the precondition for a reentrant call to runAll, although it can still call getRuns which imposes no precondition on *inv*. After the update

```
void runAll() {    assert self.inv = Queue && ! self.com;
  unpack self from Queue;
  Qnode p := self.tsks; int i := 0;
  while p ≠ null do {
    if p.getCount() < p.getLimit() then p.run(); i := i+1; fi; p := p.getNxt(); }
  self.runs := self.runs + i;
  pack self as Queue; }
void add(Task t, int lim){    assert self.inv = Queue && ! self.com;
  unpack self from Queue;
  Qnode n := new Qnode; setown n to (self,Queue);
  n.setNxt(tsks); n.setTsk(t,lim); self.tsks := n;
  pack self as Queue; } }
```

Fig. 3. Methods of class Queue with selected annotations

Table 1. Stipulated preconditions of field update and of the special commands

assert $e_1.inv > C$; /* where C is the class that declares f; i.e., $f \in dom(dfields\, C)$ */
$e_1.f{:=}e_2$

assert $e.inv = super C \wedge \mathscr{I}^C(e) \wedge \forall p \mid p.own = (e,C) \Rightarrow \neg p.com \wedge p.inv = type\, p$;
pack e **as** C /* sets $e.inv := C$ and $p.com := true$ for all p with $p.own = (e,C)$ */

assert $e.inv = C \wedge \neg e.com$;
unpack e **from** C /* sets $e.inv := super C$ and $p.com := false$ for all p with $p.own = (e,C)$ */

assert $e_1.inv = \mathsf{Object} \wedge (e_2 = \mathbf{null} \vee e_2.inv > C)$;
setown e_1 **to** (e_2,C) /* sets $e_1.own := (e_2,C)$ */

to runs, $\mathscr{I}^{Queue}$ holds again as required by the precondition (not shown) of **pack**. The ghost field *com* is discussed below.

In order to maintain (1) as a program invariant, it is necessary to control updates to fields on which invariants depend. The idea is that, to update field f of some object p, all objects o whose invariant depends on $p.f$ must be unpacked. Put differently, $\mathscr{I}(o)$ should depend only on state encapsulated for o. The discipline uses a form of ownership for this purpose: $\mathscr{I}(o)$ may depend only on objects transitively owned by o. For example, an instance of Queue owns the Qnodes reached from field tsks.

Ownership is embodied in an auxiliary field *own*, so that if $p.own = (o,C)$ then o directly owns p and an admissible invariant $\mathscr{I}^D(o)$ may depend on p for types D with $type(o) \leq D \leq C$. The objects transitively owned by o are called its *island*. For modular reasoning, it is not feasible to require as an explicit precondition for each field update that all transitive owners are unpacked. A third ghost field, *com*, is used to enforce a protocol whereby packing/unpacking is dynamically nested or bracketed (though this need not be textually apparent).

In addition to (1), two additional conditions are imposed as program invariants, i.e., to hold in all reachable states of all objects. The first may be read “an object is committed to its owner if its owner is packed". The second says that a committed object is fully packed. These make it possible for an assignment to $p.f$ to be subject only to the precondition $p.inv > C$ where C is the class that declares f.

The invariants are formalized in Def. 3 in Sect. 4. The stipulated preconditions appear in Table 1, which also describes the semantics of the pack and unpack statements in detail.[2] The diligent reader may enjoy completing the annotation of Fig. 3 according to the rules of Table 1. Consult [6, 21] for more leisurely introductions to the discipline.

2.2 Representation Independence

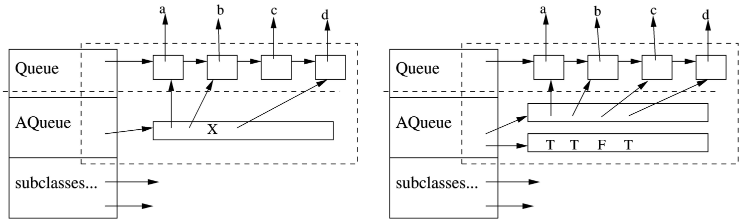
Consider the subclass AQueue of Queue declared in Fig. 4. It maintains an array, actsks, of tasks which is used in an overriding declaration of runAll intended as an optimization for the situation where many tasks are inactive (have reached their limit). We've dropped runs and getRuns for brevity. Method add exhibits a typical pattern: unpack to establish the condition in which a super call can be made (since the superclass unpacks from its own level); after that call, reestablish the current class invariant. (This imposes proof obligations on inheritance, see [6].)

The implementation of Fig. 4 does not set actsks[i] to null immediately when the task's count reaches its limit; rather, that situation is detected on the subsequent invocation of runAll. An alternative implementation is given in Fig. 5; it uses a different data structure and handles the limit being reached as soon as it happens. Both implementations maintain an array of Qnode, but in the alternative implementation, its array artsk is accompanied by a boolean array brtsk. Instead of setting entry i null when the node's task has reached its limit, brtsk[i] is set false.

We claim that the two versions are equivalent, in the context of arbitrary client programs (and subclasses, though for lack of space we do not focus on subclasses in the sequel). We would like to argue as follows. Let $filt1(o.\mathsf{actsks})$ be the sequence of non-null elements of $o.\mathsf{actsks}$ with $\mathsf{count} < \mathsf{limit}$. Let $filt2(ts, bs)$ take an array ts of tasks and a same-length array bs of booleans and return the subsequence of those tasks n in ts where bs is true and $n.\mathsf{count} < n.\mathsf{limit}$. Consider the following relation that connects a state for an instance o of the original implementation (Table 4) with an instance o' for the alternative: $filt1(o.\mathsf{actsks}) = filt2(o'.\mathsf{artsk}, o'.\mathsf{brtsk})$. The idea is that methods of the new version behave the same as the old version, modulo this change of representation. That is, for each method of AQueue, parallel execution of the two versions from a related pair of states results in a related pair of outcomes. (For this to hold we need to conjoin to the relation the invariants associated with the two versions, e.g., the second version requires artsk.length=brtsk.length.)

The left side of the picture below is an instance of some subclass of AQueue, sliced into the fields of Queue, AQueue, and subclasses; dashed lines show the objects encapsulated at the two levels relevant to reasoning about AQueue —namely the Qnodes reached from tsks and the array actsks.

[2] We omit the preconditions $e \neq \textbf{null}$ and "e not error" that are needed for the rest of the precondition to be meaningful. Different verification systems make different choices in handling errors in assertions. Our formulation follows [28] and differs superficially from [6, 21].



On the right is an instance for the alternate implementation of AQueue. It is the connection between these two islands that is of interest to the programmer. The “a”. . . “d” of the figure indicate that both versions reference the same sequence of tasks, although those tasks are not part of the islands.

In general, a *local coupling* is a binary relation on islands. It relates the state of an island for one implementation of the class of interest with an island for the alternative.

A local coupling gives rise to an *induced coupling* relation on the complete program state: Two heaps are related by the induced coupling provided that (a) they can be partitioned into islands and (b) the islands can be put into correspondence so that each corresponding pair is related by the local coupling. Moreover, the remaining objects (not in an island) are related by equality. (More precisely, equality modulo a bijection on locations, to take into account differences in allocation between the two versions.) The details are not obvious and are formalized later.

The point of the abstraction theorem is to justify that it is sufficient to check that the induced coupling is preserved by methods of AQueue, assuming the changed data structure is encapsulated and can neither affect nor be affected by client programs. At first glance one might expect the proof obligation to be that each method of AQueue preserves the local coupling, and indeed this will be the focus of reasoning in practice. But in general a method may act on more than just the island for self, e.g., by invoking

```
class AQueue extends Queue {
  private Qnode[ ] actsks;   private int  alen;
  void  add(Task t, int  lim) {    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    super.add(t,lim); actsks[alen] := self.tsks; self.alen := self.alen+1;
    pack self as AQueue; }
  void  runAll() {   assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    int  i := self.alen - 1;
    while i ≥ 0 do  {
      Qnode qn := self.actsks[i];
      if  qn ≠ null then  if  qn.getCount() < qn.getLimit()
                          then  qn.run(); else  self.actsks[i] := null; fi; fi;
      i := i - 1; }
    pack self as AQueue; } }
```

Fig. 4. First version of Class AQueue. An invariant: actsks[0..alen-1] contains any n in tsks with n.count < n.limit, in reverse order. (There may also be nulls and some n with n.count = n.limit). The elided constructor allocates actsks and we ignore the issue of the array becoming full

```
class AQueue extends Queue {
  private Qnode[ ] artsk;
  private boolean[ ] brtsk;
  private int len;
  void add(Task t, int lim) {
    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    super.add(t,lim); self.artsk[alen] := self.tsks; self.brtsk[len] := true; self.len := len+1;
    pack self as AQueue; }
  void runAll() {
    assert self.inv= AQueue && ! self.com;
    unpack self from AQueue;
    int i := self.len - 1;
    while i ≥ 0 do {
      if self.brtsk[i] then Qnode n := self.artsk[i]; int diff := n.limit - n.count;
                            if diff ≤ 1 then self.brtks[i] := false; fi;
                            if diff ≠ 0 then n.run(); fi; fi;
      i := i - 1; }
    pack self as AQueue; } }
```

Fig. 5. Alternative implementation of AQueue

methods on client objects or on other instances of AQueue. So the proof obligation is formalized in terms of the induced coupling.

In fact the proof obligation is not simply that each corresponding pair of method implementations preserves the coupling, but rather that they preserve the coupling *under the assumption that any method they invoke preserves the coupling*.[3] There is also a proof obligation for initialization but it is straightforward so we do not discuss it in connection with the examples.

For example, in the case of method runAll, one must prove that the implementations given in Fig. 4 and in Fig. 5 preserve the coupling on the assumption that the invoked methods getCount, getLimit, Qnode.run, etc. preserve the coupling. The assumption is not so important for getCount or getLimit. For one thing, it is possible to fully describe their simple behavior. For another, the alternative implementation of runAll does not even invoke these methods but rather accesses the fields directly.

The assumption about Qnode.run is crucial, however. Because run invokes, in turn, Task.run, essentially nothing is known about its behavior. For this reason both implementations of runAll invoke run on the same tasks in the same order; otherwise, it is hard to imagine how equivalence of the implementations could be verified in a modular way, i.e., reasoning only about class AQueue. But here we encounter the problem with simulation based reasoning that is analogous to the problem with invariants and reentrant callbacks. There is no reason for the coupling to hold at intermediate points of the methods of AQueue. If a method is invoked at such a point, the assumption that the

[3] The reason this is sound is similar to the justification for proof rules for recursive procedures: it is essentially the induction step for a proof by induction on the maximum depth of the method call stack.

called method preserves the coupling is of no use —just as the assumption of invariant-preservation is of no use if a method is invoked in a state where the invariant does not hold.

The Boogie discipline solves the invariant problem for an object o by replacing the declared invariant $\mathscr{I}(o)$ with an implication —see (1)— that is true in all states. As with invariants, so too with couplings: It does not make sense to ask a coupling to hold in every state, because two different implementations with nontrivial differences do not have lockstep correspondence of states. (For example, imagine that in the alternative version, the arrays are compressed every 100th invocation of runAll.) Our generalization of the Boogie idea is that the local coupling relation for a particular (pair of) island(s) is conditioned on an *inv* field so that the local coupling may hold in *some pairs of states* at intermediate points —in particular, at method calls that can lead to reentrant callbacks.

Consider corresponding instances o, o' of the two versions of AQueue. The local coupling serves to describe the corresponding pair of islands when o and o' are packed. So the induced coupling relation on program states requires corresponding pairs of islands to satisfy the local coupling just when they are packed. Because *inv* is part of the behavior observable at the level of reasoning, we can assume both versions follow the same pattern of packing (though not necessarily of control structure) and thus include $o.inv = o'.inv$ as a conjunct of the induced coupling.

Consider the two implementations of runAll. To a first approximation, what matters is that each updates some internal state and then both reach a point where run is invoked. At that point, the *local* coupling does not hold —but the *induced* coupling relation can and does hold, because the island is unpacked. This parallels the way $\mathscr{I}^C(o)$ can be false while $o.inv \leq C \Rightarrow \mathscr{I}^C(o)$ remains true, recall (1). So we can use the assumption about called methods to conclude that the coupling holds after the corresponding calls to run.

The hardest part of the proof for runAll is at the point where the two implementations pack self to *AQueue*. Just as both implementations invoke run (and on the same queue nodes), both need to pack in order to preserve the coupling. And at this point we have to argue that the local coupling is reestablished. To do so, we need to know the state of the internal structures that have been modified. We would like to argue that the only modifications are only those explicit in the code of runAll, but what about the effect of run? Owing to the preconditions on add and runAll, the only possible reentrant callbacks are to getRuns and this does no updates. (In other examples, modifies specifications would be needed at this point for modular reasoning.)

This concludes the sketch of how our abstraction theorem handles reentrant callbacks and encapsulation using the inv/own discipline. Several features of the discipline need to be adapted, in ways which also make sense in terms of informal considerations of information hiding. The additional restrictions are formalized in Section 5 and their significance discussed in Section 7. As a preview we make the following remarks, using "*Abs*" as the generic name for a class for which two versions are considered.

The discipline does not constrain field access, as reading cannot falsify an invariant predicate. Of course for reasons of information hiding one expects that visibility and alias confinement are used to prevent most or all reads of encapsulated objects. Information hiding is exactly what is formalized by representation independence and indeed the abstraction theorem fails if a client can read fields of encapsulated objects. So every field access $e.f$ is subject to a precondition: If e is transitively owned by some instance o of the class, *Abs*, under revision, then either self is o or else self is transitively owned by o.

Another problematic feature is that "**pack** e **as** C" can occur in any class, so long as its preconditions are established. This means that, unlike traditional theories, an invariant is not simply established at initialization. In our theory the local coupling must be established preceding each "**pack** e **as** *Abs*". We aim for modular reasoning where only *Abs* needs to be considered, so we insist that **pack** e **as** C with $C = Abs$ occurs only in code of *Abs*.

Although the discipline supports hierarchical ownership, our technical treatment benefits from heap partitioning ideas from separation logic (we highlight the connections where possible, e.g., in Proposition 1). For this reason and a more technical one, it is convenient to prevent an instance of *Abs* from transitively owning another instance of *Abs* (lest their islands be nested). This can be achieved by a simple syntactic restriction. It does not preclude that, say, class AQueue can hold tasks that own AQueue objects, because an instance of AQueue owns its representation objects (the Qnodes), not the tasks they contain. Nor does it preclude hierarchical ownership, e.g., *Abs* could own a hashtable that in turn owns some arrays.

Finally, consider ownership transfer across the encapsulation boundary. The hardest case is where a hitherto-encapsulated object is released to a client, e.g., when a memory manager allocates nodes from a free list [29, 4]. This can be seen as a deliberate exposure of representation and thus is observable behavior that must be retained in a revised version of the abstraction. Yet encapsulated data of the two versions can be in general quite different. To support modular reasoning about the two versions, it appears essential to restrict outward transfer of objects encapsulated for *Abs* to occur only in code of *Abs*, where the reasoner can show that the coupling is preserved.

3 An Illustrative Language

Following [6, 21], we formalize the *inv*/*own* discipline in terms of a language in which fields have public visibility, to illuminate the conditions necessary for sound reasoning about invariants and simulations. In practice, private and protected visibility and perhaps lightweight alias control would serve to automatically check most of the conditions. This section formalizes the language, adapting notations and typing rules from Featherweight Java [19] and imperative features and the special commands from our previous papers [2, 28].

A complete program is given as a *class table*, CT, that maps class name C to a declaration $CT(C)$ of the form **class** C **extends** D $\{\ \bar{T}\ \bar{f};\ \bar{M}\ \}$. The categories T, M are given by the grammar in Table 2. Barred identifiers like $\bar{T}$ indicate finite lists, e.g., $\bar{T}\ \bar{f}$ stands for a list $\bar{f}$ of field names with corresponding types $\bar{T}$.

Well formed class tables are characterized using typing rules which are expressed using some auxiliary functions that in turn depend on the class table, allowing classes to make mutually recursive references to other classes, without restriction. In particular, this allows recursive methods (so we omit loops). For a class C, *fields*(C) is defined as the inherited and declared fields of C; *dfields*(C) is the fields declared in C; *super*(C) is the direct superclass of C. For a method declaration, $T\ m(\bar{T}_1\ \bar{x})\ \{S\}$ in C, the method type *mtype*(m, C) is $\bar{T}_1 \rightarrow T$ and parameter names, *pars*(m, C), is $\bar{x}$. For m inherited

Table 2. Grammar

$C \in \mathit{ClassName} \quad m \in \mathit{MethName} \quad f \in \mathit{FieldName} \quad x, \mathsf{self}, \mathsf{result} \in \mathit{VarName}$

$T \ ::= \ \mathbf{bool} \mid \mathbf{void} \mid C$	data type
$M \ ::= \ T\ m(\bar{T}\ \bar{x})\ \{S\}$	method declaration
$S \ ::= \ x{:=}e \mid e.f{:=}e$	assign to local var. or param., update field
$\mid \ x{:=}\,\mathbf{new}\ C \mid x{:=}\,e.m(\bar{e})$	object creation, method call
$\mid \ T\ x{:=}\,e\ \mathbf{in}\ S \mid S;\ S \mid \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S\ \mathbf{fi}$	local variable, sequence, conditional
$\mid \ \mathbf{pack}\ e\ \mathbf{as}\ C \mid \mathbf{unpack}\ e\ \mathbf{from}\ C$	set *inv* to C, set *inv* to $superC$
$\mid \ \mathbf{setown}\ e\ \mathbf{to}\ (e',C)$	set $e.own$ to (e',C)
$\mid \ \mathbf{assert}\ \mathscr{P}$	assert (semantic predicate $\mathscr{P}$)
$e \ ::= \ x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false}$	variable, constant
$\mid \ e.f \mid e = e \mid e\ \mathbf{is}\ C \mid (C)\ e$	field access, ptr. equality, type test, cast

Table 3. Typing rules for selected expressions and commands

$$\frac{\Gamma \vdash e : C \qquad (f : T) \in \mathit{fields}(C)}{\Gamma \vdash e.f : T} \qquad \frac{\Gamma \vdash e_1 : D_1 \qquad \Gamma \vdash e_2 : D_2 \qquad D_2 \leq C}{\Gamma \vdash \mathbf{setown}\ e_1\ \mathbf{to}\ (e_2, C)}$$

$$\frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash \mathbf{pack}\ e\ \mathbf{as}\ C} \qquad \frac{\Gamma \vdash e : D \quad D \leq C}{\Gamma \vdash \mathbf{unpack}\ e\ \mathbf{from}\ C}$$

$$\frac{\Gamma \vdash e : D \qquad \mathit{mtype}(m,D) = \bar{T} \to U \qquad x \neq \mathsf{self} \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U} \leq \bar{T} \qquad U \leq \Gamma x}{\Gamma \vdash x{:=}\,e.m(\bar{e})}$$

in C, $\mathit{mtype}(m,C) = \mathit{mtype}(m,D)$ and $\mathit{pars}(m,C) = \mathit{pars}(m,D)$ where D is the direct superclass of C.

For use in the semantics, $\mathit{xfields}(C)$ extends $\mathit{fields}(C)$ by assigning "types" to the auxiliary fields: $com : \mathbf{bool}$, $own : \mathsf{owntyp}$, and $inv : (\mathsf{invtyp}C)$. (These are not included in *FieldName*.) Neither $\mathsf{invtyp}C$ nor owntyp are types in the programming language but there are corresponding semantic domains and the slight notational abuse is convenient.

A *typing context* Γ is a finite function from variable names to types, such that $\mathsf{self} \in \mathit{dom}\,\Gamma$. Selected typing rules for expressions and commands are given in Table 3. A judgement of the form $\Gamma \vdash e : T$ says that expression e has type T in the context of a method of class $\Gamma\mathsf{self}$, with parameters and local variables declared by Γ. A judgement $\Gamma \vdash S$ says that S is a command in the same context. A class table CT is well formed if each method declaration $M \in CT(C)$ is well formed in C; this is written $C \vdash M$ and defined by the following rule:

$$\frac{\begin{array}{c}\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S \\ \text{if } \mathit{mtype}(m, superC) \text{ is defined then } \mathit{mtype}(m, superC) = \bar{T} \to T \text{ and } \mathit{pars}(m, superC) = \bar{x}\end{array}}{C \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

To formalize assertions, we prefer to avoid both the commitment to a particular formula language and the complication of an environment for declaring predicate names to be interpreted in the semantics. So we indulge in a mild and commonplace abuse of notation: the syntax of **assert** uses a semantic predicate. We say $\Gamma \vdash \mathbf{assert}\ \mathscr{P}$ is well

Table 4. Semantic categories θ and domains $\llbracket\theta\rrbracket$. (Readers familiar with notation for dependent function spaces might prefer to write $\llbracket\mathsf{pre\text{-}heap}\rrbracket = (o : Loc \nrightarrow \llbracket\mathsf{state}\,(type\ o)\rrbracket)$ and similarly for $\llbracket\mathsf{state}\,C\rrbracket$ and $\llbracket\Gamma\rrbracket$.)

$$\begin{array}{ll}
\theta ::= T \mid \Gamma \mid \theta_\perp & \\
\quad \mid\ \mathsf{owntyp} \mid \mathsf{invtyp}\,C \mid \mathsf{state}\,C & \textit{own} \text{ and } \textit{inv} \text{ val., object state} \\
\quad \mid\ \mathsf{pre\text{-}heap} \mid \mathsf{heap} \mid \mathsf{heap} \otimes \Gamma \mid \mathsf{heap} \otimes T & \text{heap fragment, closed heap, state, result} \\
\quad \mid\ (\Gamma \vdash \mathsf{cmd}) \mid (\Gamma \vdash T) \mid (C, \bar{x}, \bar{T} \to T_1) \mid \mathsf{menv} & \text{command, expr., method, method envir.}
\end{array}$$

$$\llbracket C\rrbracket = \{nil\} \cup \{o \in Loc \mid type\,o \le C\} \qquad \llbracket\mathbf{bool}\rrbracket = \{true, false\} \qquad \llbracket\mathbf{void}\rrbracket = \{it\}$$

$$\begin{array}{ll}
\llbracket\mathsf{invtyp}\,C\rrbracket & = \{B \mid C \le B\} \\
\llbracket\mathsf{owntyp}\rrbracket & = \{(o, C) \mid o = nil \vee type\,o \le C\} \\
\llbracket\mathsf{state}\,C\rrbracket & = \{s \mid dom\,s = dom(xfields\,C) \wedge \forall (f : T) \in xfields\,C \mid s\,f \in \llbracket T\rrbracket\} \\
\llbracket\mathsf{pre\text{-}heap}\rrbracket & = \{h \mid dom\ h \subseteq_{fin} Loc \wedge \forall o \in dom\ h \mid h\ o \in \llbracket\mathsf{state}\,(type\ o)\rrbracket\} \\
\llbracket\mathsf{heap}\rrbracket & = \{h \mid h \in \llbracket\mathsf{pre\text{-}heap}\rrbracket \wedge \forall s \in rng\ h \mid rng\ s \cap Loc \subseteq dom\ h\} \\
\llbracket\Gamma\rrbracket & = \{s \mid dom\,s = dom\,\Gamma \wedge s\,\mathsf{self} \neq nil \wedge \forall x \in dom\,s \mid s\,x \in \llbracket\Gamma\,x\rrbracket\ \} \\
\llbracket\mathsf{heap} \otimes \Gamma\rrbracket & = \{(h, s) \mid h \in \llbracket\mathsf{heap}\rrbracket \wedge s \in \llbracket\Gamma\rrbracket \wedge rng\ s \cap Loc \subseteq dom\ h\} \\
\llbracket\mathsf{heap} \otimes T\rrbracket & = \{(h, v) \mid h \in \llbracket\mathsf{heap}\rrbracket \wedge v \in \llbracket T\rrbracket \wedge (v \in Loc \Rightarrow v \in dom\ h)\} \\
\llbracket\Gamma \vdash \mathsf{cmd}\rrbracket & = \llbracket\mathsf{heap} \otimes \Gamma\rrbracket \to \llbracket(\mathsf{heap} \otimes \Gamma)_\perp\rrbracket \\
\llbracket\Gamma \vdash T\rrbracket & = \{v \mid v \in (\llbracket\mathsf{heap} \otimes \Gamma\rrbracket \to \llbracket T\rrbracket_\perp) \wedge \forall h, s \mid v(h, s) \in Loc \Rightarrow v(h, s) \in dom\,h\} \\
\llbracket(C, \bar{x}, \bar{T} \to T_1)\rrbracket & = \llbracket\mathsf{heap} \otimes (\bar{x} : \bar{T}, \mathsf{self} : C)\rrbracket \to \llbracket(\mathsf{heap} \otimes T_1)_\perp\rrbracket \\
\llbracket\mathsf{menv}\rrbracket & = \{\mu \mid \forall C, m \mid \mu C m \text{ is defined iff } mtype(m, C) \text{ is defined,} \\
 & \qquad\quad \text{and } \mu C m \in \llbracket C, pars(m, C), mtype(m, C)\rrbracket \text{ if } \mu C m \text{ defined } \}
\end{array}$$

formed provided that $\mathscr{P}$ is a set of program states for context Γ. This treatment of assertions is also convenient for taking advantage of a theorem prover's native logic.

Semantics. Some semantic domains correspond directly to the syntax. For example, each data type T denotes a set $\llbracket T\rrbracket$ of values. The meaning of context Γ is a set $\llbracket\Gamma\rrbracket$ of stores; a *store* $s \in \llbracket\Gamma\rrbracket$ is a type-respecting assignment of locations and primitive values to the local variables and parameters given by a typing context Γ. The semantics, and later the coupling relation, is structured in terms of category names θ given in Table 4 which also defines the semantic domains.

A *program state* for context Γ is a pair (h, s) where s is in $\llbracket\Gamma\rrbracket$ and h is a *heap*, i.e., a finite partial function from locations to object states. An *object state* is a type-respecting mapping of field names to values. A command typable in Γ denotes a function mapping each program state (h, s) either to a final state (h_0, s_0) or to the distinguished value $\perp$ which represents runtime errors, divergence, and assertion failure. An *object state* is a mapping from (extended) field names to values. A *pre-heap* is like a heap except for possibly having dangling references. If h, h' are pre-heaps with disjoint domains then we write $h * h'$ for their union; otherwise $h * h'$ is undefined. Function application associates to the left, so $h\,o\,f$ is the value of field f of the object $h\,o$ at location o. We also write $h\,o.f$. Application binds more tightly than binary operator symbols and ",".

We assume that a countable set *Loc* is given, along with a distinguished value *nil* not in *Loc*. We assume given a function *type* from *Loc* to non-primitive types distinct from Object, such that for each C there are infinitely many locations o with $type\ o = C$. This is used in a way that is equivalent to tagging object states with their type.

Table 5. Semantics of selected expressions and commands. To streamline the treatment of $\bot$, the metalanguage expression "let $\alpha = \beta$ in $\ldots$" denotes $\bot$ if β is $\bot$. We use function extension notation $[h \mid o \mapsto st]$ for h extended or overridden at o with value st. For brevity the nested function extension for field update is written $[h \mid o.f \mapsto v]$

$$
\begin{array}{ll}
\llbracket \Gamma \vdash e.f : T \rrbracket(h,s) & = \mathsf{let}\ o = \llbracket \Gamma \vdash e : C \rrbracket(h,s)\ \mathsf{in\ if}\ o = nil\ \mathsf{then}\ \bot\ \mathsf{else}\ h\,o.f \\
\llbracket \Gamma \vdash x{:=}e.m(\bar{e}) \rrbracket \mu(h,s) & = \mathsf{let}\ o = \llbracket \Gamma \vdash e : T \rrbracket(h,s)\ \mathsf{in\ if}\ o = nil\ \mathsf{then}\ \bot\ \mathsf{else} \\
& \quad \mathsf{let}\ \bar{v} = \llbracket \Gamma \vdash \bar{e} : \bar{U} \rrbracket(h,s)\ \mathsf{in\ let}\ \bar{x} = pars(m,T)\ \mathsf{in} \\
& \quad \mathsf{let}\ s_1 = [\bar{x} \mapsto \bar{v}, \mathsf{self} \mapsto o]\ \mathsf{in} \\
& \quad \mathsf{let}\ (h_1, v_1) = \mu(type\,o)m(h,s_1)\ \mathsf{in}\ (h_1, [s \mid x \mapsto v_1]) \\
\llbracket \Gamma \vdash \mathbf{assert}\ \mathscr{P} \rrbracket \mu(h,s) & = \mathsf{if}\ (h,s) \in \mathscr{P}\ \mathsf{then}\ (h,s)\ \mathsf{else}\ \bot \\
\llbracket \Gamma \vdash \mathbf{pack}\ e\ \mathbf{as}\ C \rrbracket \mu(h,s) = & \\
\quad \mathsf{let}\ q = \llbracket \Gamma \vdash e : D \rrbracket(h,s)\ \mathsf{in\ if}\ q = nil\ \mathsf{then}\ \bot\ \mathsf{else} & \\
\quad \mathsf{let}\ h_1 = \lambda p \in dom\ h \mid \mathsf{if}\ h\,p.own = (q,C)\ \mathsf{then}\ [h\,p \mid com \mapsto true]\ \mathsf{else}\ h\,p\ \mathsf{in}\ ([h_1 \mid q.inv \mapsto C],\ s) & \\
\llbracket \Gamma \vdash \mathbf{unpack}\ e\ \mathbf{from}\ C \rrbracket \mu(h,s) = & \\
\quad \mathsf{let}\ q = \llbracket \Gamma \vdash e : N \rrbracket(h,s)\ \mathsf{in\ if}\ q = nil\ \mathsf{then}\ \bot\ \mathsf{else} & \\
\quad \mathsf{let}\ h_1 = \lambda p \in dom\ h \mid \mathsf{if}\ h\,p.own = (q,C)\ \mathsf{then}\ [h\,p \mid com \mapsto false]\ \mathsf{else}\ h\,p\ \mathsf{in} & \\
\quad ([h_1 \mid q.inv \mapsto super\,C],\ s) & \\
\llbracket \Gamma \vdash \mathbf{setown}\ e_1\ \mathbf{to}\ (e_2, C) \rrbracket \mu(h,s) = & \\
\quad \mathsf{let}\ q = \llbracket \Gamma \vdash e_1 : N_1 \rrbracket(h,s)\ \mathsf{in\ if}\ q = nil\ \mathsf{then}\ \bot\ \mathsf{else} & \\
\quad \mathsf{let}\ p = \llbracket \Gamma \vdash e_2 : N_2 \rrbracket(h,s)\ \mathsf{in}\ ([h \mid q.own \mapsto (p,C)],\ s) &
\end{array}
$$

The meaning of a derivable command typing $\Gamma \vdash S$ will be defined to be a function sending each method environment μ to an element of $\llbracket \Gamma \vdash \mathsf{cmd} \rrbracket$. (The keyword "cmd" just provides notation for command meanings.) That is, $\llbracket \Gamma \vdash S \rrbracket \mu$ is a state transformer $\llbracket \mathsf{heap} \otimes \Gamma \rrbracket \to \llbracket (\mathsf{heap} \otimes \Gamma)_\bot \rrbracket$. The method environment is used only to interpret the method call command. Meanings for expressions and commands are defined, in Table 5, by recursion on typing derivation. The semantics is defined for an arbitrary location-valued function *fresh* such that $type(fresh(C,h)) = C$ and $fresh(C,h) \notin dom\,h$.

The meaning of a well typed method declaration M, of the form $M = T\ m(\bar{T}\ \bar{x})\{S\}$, is the total function in $\llbracket \mathsf{menv} \rrbracket \to \llbracket (C, \bar{x}, \bar{T} \to T) \rrbracket$ defined as follows: Given a method environment μ, a heap h and a store $s \in \llbracket \bar{x} : \bar{T}, \mathsf{result} : C \rrbracket$, first execute S to obtain the updated heap h_0 and the updated store s_0; then return $(h_0, s_0(\mathsf{result}))$. A method environment μ maps each C, m to a meaning obtained in this way or by inheritance. For well formed class table CT, the semantics $\llbracket CT \rrbracket$ is defined as the least upper bound of an ascending chain of method environments—the approximation chain—with method declarations interpreted as above and a suitable interpretation for inherited methods. Details omitted.

A *predicate* for state type Γ is just a subset $\mathscr{P} \subseteq \llbracket \mathsf{heap} \otimes \Gamma \rrbracket$. For emphasis we can write $(h,s) \models \mathscr{P}$ for $(h,s) \in \mathscr{P}$. Note that $\bot \notin \mathscr{P}$. We give no formal syntax to denote predicates but rather use informal metalanguage for which the correspondence should be clear. For example, "$\mathsf{self}.f \neq$ **null**" denotes the set of (h,s) with $h(s\,\mathsf{self}).f \neq nil$. and "$\forall o \mid \mathscr{P}(o)$" denotes the set of (h,s) such that $(h,s) \models \mathscr{P}(o)$ for all $o \in dom\ h$. Note that quantification over objects (e.g., in Table 1 and Def. 3) is interpreted to mean quantification over allocated locations; the range of quantification can include unreachable objects but this causes no problems.

By contrast with [6, 21], we have taken care to separate the annotations required by the *inv*/*own* discipline from the semantics of commands. The invariants encoded in the semantic domains (e.g., the value in a field has its declared type and there are no dangling pointers) depend in no way on assertions, only on typing. A similar semantic model has been machine checked in PVS [27].

4 The *inv*/*own* Discipline

The discipline reviewed in Sect. 2.1 is designed to make (1) a program invariant for every object. This is achieved using additional program invariants that govern ownership. We formalize this as a global predicate, *disciplined*, defined in three steps.

Definition 1 (transitive C- and $C\uparrow$-ownership). For any heap h, the relation $o \succ^h_C p$ on *dom h*, read "*o owns p at C in h*", holds iff either $(o,C) = h\,p.own$ or there are q and D such that $(o,C) = h\,q.own$ and $q \succ^h_D p$. The relation $o \succ^h_{C\uparrow} p$ holds iff there is some D with $C \leq D$ and $o \succ^h_D p$.

Definition 2 (admissible invariant). A predicate $\mathscr{P} \subseteq [\![\mathsf{heap} \otimes (\mathsf{self} : C)]\!]$ is *admissible as an invariant for C* provided that it is not falsifiable by creation of new objects and for every (h,s) and o, f such that $\mathscr{P}$ depends on $o.f$ in (h,s), field f is neither *inv* nor *com*, and one of the following conditions holds: $o = s(\mathsf{self})$ and f is in $dom(xfields\,C)$ or $s(\mathsf{self}) \succ^h_{C\uparrow} o$.

For dependence on fields of self, the typing condition, $f \in dom(xfields\,C)$, prevents an invariant for C from depending on fields declared in a subclass of C (which could be expressed in a formula using a cast). An invariant can depend on any fields of objects owned at C or above. We refrain from introducing syntax for declaring invariants. In the subsequent definitions, an admissible invariant $\mathscr{I}^C$ is assumed given for every class C. We assume $\mathscr{I}^{Object} = \mathbf{true}$.

Definition 3 (disciplined, $\mathscr{J}$). A heap h is *disciplined* if $h \models \mathscr{J}$ where $\mathscr{J}$ is defined to be the conjunction of the following: $\forall o, C \mid o.inv \leq C \Rightarrow \mathscr{I}^C(o)$

$$\forall o, C, p \mid o.inv \leq C \wedge p.own = (o,C) \Rightarrow p.com$$
$$\forall o \mid o.com \Rightarrow o.inv = type(o)$$

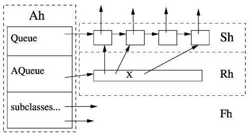
A state (h,s) is *disciplined* if h is. Method environment μ *is disciplined* provided that every method maintains $\mathscr{J}$ (i.e., for any C, m, h, s, if $h \in \mathscr{J}$ and $\mu\,C\,m\,(h,s) = (h_0, v)$— and thus $\mu\,C\,m\,(h,s) \neq \bot$— then $h_0 \in \mathscr{J}$).

Lemma 1 (transitive ownership). Suppose h is disciplined and $o \succ^h_C p$. Then (a) $type\,o \leq C$ and (b) $h\,o.inv \leq C$ implies $h\,p.com = true$.

Corollary 1. If h is disciplined, $o \succ^h_C p$, and $h\,p.inv > type\,p$ then $h\,o.inv > C$.

Partitioning the Heap. We partition the objects in the heap in order to formalize the encapsulation boundary depicted in Sect. 2.2. Given an object $o \in dom\,h$ and class name

A with $\mathit{type}\, o \leq A$ we can partition h into pre-heaps Ah (the A-object), Rh (the representation of o for class A), Sh (objects owned by o at a superclass), and Fh (free from o) determined by the following conditions: Ah is the singleton $[o \mapsto h\, o]$, Rh is h restricted to the set of p with $o \succ^h_A p$, Sh is h restricted to the set of p with $o \succ^h_C p$ for some $C > A$, and Fh is the rest of h. Note that if $o \succ^h_B p$ for some proper subclass $B < A$ then $p \in \mathit{dom}\, Fh$. A pre-heap of the form $Ah * Rh * Sh$ is called an *island*. In these terms, dependency of admissible invariants is described in the following Proposition. As an illustration, here is the island for the left side of the situation depicted in Sect. 2.2:



Proposition 1 (island). Suppose $\mathscr{I}^C$ is an admissible invariant for C and $o \in \mathit{dom}\, h$ with $\mathit{type}\, o \leq C$. If $h = Fh * Ah * Rh * Sh$ is the partition defined above then $Fh_0 * Ah * Rh * Sh \models \mathscr{I}^C(o)$ iff $h \models \mathscr{I}^C(o)$, for all Fh_0 such that $Fh_0 * Ah * Rh * Sh$ is a heap.

The Discipline. To impose the stipulated preconditions of Table 1 we consider programs with the requisite syntactic structure (similar to formal proof outlines).

Definition 4 (properly annotated). The *annotated commands* are the subset of the category of commands where each **pack**, **unpack**, **setown**, and field update is immediately preceded by an **assert**. A *properly annotated command* is an annotated command such that each of these assertions is (or implies) the precondition stipulated in Table 1. A *properly annotated class table* is one such that each method body is properly annotated.

For any class table and family of invariants there exists a proper annotation: just add **assert** commands with the stipulated preconditions. For practical interest, of course, one wants assertions that can collectively be proved correct. The abstraction theorem depends on proper annotation but does not depend on the invariants themselves; one may take $\mathscr{I}^C = \textbf{true}$ for all C. What matters is ownership structure and the use of *inv*. We use the following [6, 21, 28].

Proposition 2. If method environment μ is disciplined then any properly annotated command S maintains $\mathscr{J}$ in the sense that for all (h,s), if $h \models \mathscr{J}$ and $(h_0,s_0) = [\![\Gamma \vdash S]\!]\mu(h,s)$ then $h_0 \models \mathscr{J}$. If CT is a properly annotated class table then the method environment $[\![CT]\!]$ is disciplined.

5 The Abstraction Theorem

5.1 Comparing Class Tables

We compare two implementations of a designated class *Abs*, in the context of a fixed but arbitrary collection of other classes, such that both implementations give rise to a well formed class table. The two versions can have completely different declarations, so long as methods of the same signatures are present — declared or inherited — in both. To simplify the additional precondition needed for reading fields, we consider programs desugared into a form like that used in Separation Logic.

Definition 5 (properly annotated for *Abs***).** The *annotated commands for Abs* are those of Def. 4 with the additional restriction that no expression of the form $e.f$ occurs except in commands of the form **assert** $\mathscr{P}; x{:=}\,e.f$ (in particular, no field access appears in this e). The *properly annotated commands for Abs* are those that are properly annotated according to Def. 4 and moreover

- fields of Abs have private visibility (i.e., if $f \in \mathit{dfields}\,Abs$ then accesses and updates of f only occur in code of class Abs)
- If $\Gamma\,\mathsf{self} \neq Abs$ then field access $\Gamma \vdash x{:=}\,e.f$ is subject to stipulated precondition $(\forall o \mid o \succ_{Abs} e \Rightarrow o \succ_{Abs} \mathsf{self})$
- if $\Gamma\,\mathsf{self} \neq Abs$ then $\Gamma \vdash$ **pack** e **as** Abs is not allowed
- if $\Gamma\,\mathsf{self} \neq Abs$ then $\Gamma \vdash$ **setown** e_1 **to** (e_2, C) is subject to an additional precondition: $(\exists o \mid o \succ_{Abs} e_1) \Rightarrow C = Abs \vee (\exists o \mid o \succ_{Abs} e_2)$

The effect of the last precondition is that if e_1 is initially owned at Abs then after a transfer (that occurs in code outside class Abs) it is still owned at Abs.

In order to work with heap partitions, along the lines of Prop. 1, it is convenient to have notation to extract the one object in a singleton heap. We define *pickdom* by $\mathit{pickdom}(h) = o$ where $\mathit{dom}\,h = \{o\}$; it is undefined if $\mathit{dom}\,h$ is not a singleton.

Prop. 1 considers a single object together with its owned representation; now we consider all objects of a given class.

Definition 6 (*A*-decomposition). For any class A and heap h, the *A-decomposition* of h is the set $Fh, Ah_1, Rh_1, Sh_1 \ldots, Ah_k, Rh_k, Sh_k$ (for some $k \geq 0$) of pre-heaps, all subsets of h, determined by the following conditions:

- each $\mathit{dom}Ah_i$ contains exactly one object o and $\mathit{type}\,o \leq A$
- every $o \in \mathit{dom}\,h$ with $\mathit{type}\,o \leq A$ occurs in $\mathit{dom}Ah_i$ for some i;
- $\mathit{dom}\,Rh_i = \{p \mid o \succ^h_A p\}$ where $\mathit{pickdom}Ah_i = o$;
- $\mathit{dom}\,Sh_i = \{p \mid o \succ^h_{(\mathit{super}A)\uparrow} p\}$ with $\mathit{pickdom}Ah_i = o$;
- $\mathit{dom}Fh = \mathit{dom}\,h - (\cup i \mid \mathit{dom}(Ah_i * Rh_i * Sh_i))$

We say that *no A-object owns an A-object in h* provided for every o, p in $\mathit{dom}\,h$ if $\mathit{type}\,o \leq A$ and $o \succ^h_{(\mathit{type}\,o)\uparrow} p$ then $\mathit{type}\,p \nleq A$. Def. 8 in the sequel imposes a syntactic restriction to maintain this property as an invariant, where A is the class for which two representations are compared. A consequence is that there is a unique decomposition of the heap into separate islands of the form $Ah * Rh * Sh$. We use the term “partition” even though some blocks can be empty.

Lemma 2 (*A*-partition). Suppose no A-object owns an A-object in h. Then the A-decomposition is a partition of h, that is, $h = Fh * Ah_1 * Rh_1 * Sh_1 * \ldots * Ah_k * Rh_k * Sh_k$.

To maintain the invariant that no Abs-object owns an Abs-object, we formulate a mild syntactic restriction expressed using a static approximation of ownership.

Definition 7 (may own, $\succ^{\exists}$). Given well formed CT, define $\succ^{\exists}$ to be the least transitively closed relation such that

- $D_2 \succ^{\exists} D_1$ for every occurrence of **setown** e_1 **to** (e_2, D) in a method of CT, with static types $e_1 : D_1$ and $e_2 : D_2$
- if $C \succ^{\exists} D$, $C' \leq C$ and $D' \leq D$ then $C' \succ^{\exists} D'$

If $Abs \not\succ^{\exists} Abs$ then it is a program invariant that no *Abs*-object owns an *Abs*-object (recall the definition preceding Lemma 2). This is a direct consequence of the following.

Lemma 3. It is a program invariant that if $o \succ^{h}_{C} p$ then $type\, o \succ^{\exists} type\, p$.

Definition 8 (comparable class tables). Well formed class tables CT and CT' are *comparable* with respect to class name *Abs* ($\neq$ Object) provided the following hold.

- $CT(C) = CT'(C)$ for all $C \neq Abs$.
- $CT(Abs)$ and $CT'(Abs)$ declare the same methods with the same signatures and the same direct superclass.
- For every method m declared in $CT(Abs)$, m is declared in $CT'(Abs)$ and has the same signature; *mutatis mutandis* for m declared in CT'.
- CT and CT' are properly annotated for *Abs*.
- $Abs \not\succ^{\exists} Abs$ in both CT and CT'

The last condition ensures that the *Abs*-decomposition of any disciplined heap is a partition, by Lemmas 2 and 3. We write $\vdash, \vdash'$ for the typing relation determined by CT, CT' respectively; similarly we write $[\![-]\!], [\![-]\!]'$ for the respective semantics.

5.2 Coupling Relations

The definitions are organized as follows. A *local coupling* is a suitable relation on islands. This induces a family of *coupling relations*, $\mathscr{R}\beta\,\theta$ for each category name θ and typed bijection β. Each relation $\mathscr{R}\beta\,\theta$ is from $[\![\theta]\!]$ to $[\![\theta]\!]'$. Here β is a bijection on locations, used to connect a heap in $[\![\mathsf{heap}]\!]$ to one in $[\![\mathsf{heap}]\!]'$. The idea is that β relates all objects except those in the Rh_i or Rh'_i blocks that have never been exposed. Finally, a *simulation* is a coupling that is preserved by all methods of *Abs* and holds initially.

Definition 9. A *typed bijection* is a bijective relation, β, from *Loc* to *Loc*, such that $\beta\, o\, o'$ implies $type\, o = type\, o'$ for all o, o'. A *total bijection* on h, h' is a typed bijection with $dom\, h = dom\, \beta$ and $dom\, h' = rng\, \beta$. Finally, β *fully partitions* h, h' *for Abs* if, for all $o \in dom\, h$ (resp. $o \in dom\, h'$) with $type\, o \leq Abs$, o is in $dom\, \beta$ (resp. $rng\, \beta$).

Lemma 4 (typed bijection and *Abs*-partition). Suppose β is a typed bijection with $\beta \subseteq dom\, h \times dom\, h'$ and β fully partitions h, h' for *Abs*. If h, h' are disciplined and partition as $h = Fh * \ldots Ah_j * Rh_j * Sh_j$ and $h' = Fh' * \ldots Ah'_k * Rh'_k * Sh'_k$ then $j = k$.

Definition 10 (equivalence for *Abs* modulo bijection). For any β we define a relation $\sim_{\beta}$ for data values, object states, heaps, and stores, in Table 6.

Table 6. Value equivalence for the designated class *Abs*. The relation for heap is the same as for pre-heap. For object states, $\sim$ is independent from the declared fields of $CT(Abs)$ and $CT'(Abs)$

$o \sim_\beta o'$	in $[\![C]\!]$	$\Leftrightarrow \beta\, o\, o' \vee o = nil = o'$
$v \sim_\beta v'$	in $[\![T]\!]$	$\Leftrightarrow v = v'$ for primitive types T
$s \sim_\beta s'$	in $[\![\mathsf{state}\,C]\!]$	$\Leftrightarrow \forall (f:T) \in \mathit{xfields}\,C \mid s\,f \sim_\beta s'\,f \vee (f:T) \in \mathit{dfields}\,\mathit{Abs}$
$s \sim_\beta s'$	in $[\![\Gamma]\!]$	$\Leftrightarrow \forall x \in dom\,\Gamma \mid s\,x \sim_\beta s'\,x$
$h \sim_\beta h'$	in $[\![\mathsf{pre\text{-}heap}]\!]$	$\Leftrightarrow \forall o \in dom\,h, o' \in dom\,h' \mid \beta\, o\, o' \Rightarrow h\, o \sim_\beta h' o'$
$(h,s) \sim_\beta (h',s')$	in $[\![\mathsf{heap} \otimes \Gamma]\!]$	$\Leftrightarrow h \sim_\beta h' \wedge s \sim_\beta s'$
$v \sim_\beta v'$	in $[\![\theta_\bot]\!]$	$\Leftrightarrow v = \bot = v' \vee (v \neq \bot \neq v' \wedge v \sim_\beta v'$ in $[\![\theta]\!])$
$(o,C) \sim_\beta (o',C')$	in $[\![\mathsf{owntyp}]\!]$	$\Leftrightarrow (o = nil = o') \vee (\beta\, o\, o' \wedge C = C')$
$B \sim_\beta B'$	in $[\![\mathsf{invtyp}\,C]\!]$	$\Leftrightarrow B = B'$

Equivalence hides the private fields of *Abs*. In the identity extension lemma, it is used in conjunction with the following which hides objects owned at *Abs*.

Definition 11 (encap). Suppose no A-object owns an A-object in h. Define $\mathit{encap}\,A\,h$ to be the pre-heap $Fh * Ah_1 * Sh_1 * \ldots * Ah_k * Sh_k$ where the A-partition of h is as in Lemma 2.

The most important definition is of local coupling, which is analogous to an object invariant but is a relation on pairs of pre-heaps. In Def. 2, we take an invariant $\mathscr{I}^C$ to be a predicate (set of states) and the program invariant $\mathscr{J}$ is based on the conjunction of these predicates for all objects and types —subject to *inv*, see Def. 3). By contrast, we define a local coupling $\mathscr{L}$ in terms of pre-heaps. And we are concerned with a single class, *Abs*, rather than all C. We impose the same dependency condition as in Def. 2, but in terms of pre-heaps of the form $h = Ah * Rh * Sh$. (Recall Proposition 1.)

Definition 12 (local coupling, $\mathscr{L}$). Given comparable class tables, a *local coupling* is a function, $\mathscr{L}$, that assigns to each typed bijection β a binary relation $\mathscr{L}\,\beta$ on pre-heaps that satisfies the following. First, $\mathscr{L}\,\beta$ does not depend on *inv* or *com*. Second, $\beta \subseteq \beta_0$ implies $\mathscr{L}\,\beta \subseteq \mathscr{L}\,\beta_0$. Third, for any β, h, h', if $\mathscr{L}\,\beta\,h\,h'$ then there are locations o, o' with $\beta\,o\,o'$ and $\mathit{type}\,o \leq \mathit{Abs}$ such that the *Abs* partitions of h, h' are $h = Ah * Rh * Sh$ and $h' = Ah' * Rh' * Sh'$ with

- $\mathit{pick}\,\mathit{dom}\,Ah = o$ and $\mathit{pick}\,\mathit{dom}\,Ah' = o'$
- $o \succ^{h}_{Abs} p$ for all $p \in dom(Rh)$ and $o' \succ^{h'}_{Abs} p'$ for all $p' \in dom(Rh')$
- $o \succ^{h}_{(superAbs)\uparrow} p$ for all $p \in dom(Sh)$ and $o' \succ^{h'}_{(superAbs)\uparrow} p'$ for all $p' \in dom(Sh')$
- If $\mathscr{L}\,\beta$ depends on f then f is in $\mathit{xfields}\,\mathit{Abs}$

The first three conditions ensure that $\mathscr{L}$ relates a single island, for an object of some subtype of *Abs*, to a single island for an object of the same type. Although $\mathscr{L}$ is unconstrained for the private fields of $CT(Abs)$ and $CT'(Abs)$, it may also depend on fields inherited from a superclass of *Abs* (but not on subclass fields). The induced coupling relation, defined below, imposes the additional constraint that fields of proper sub- and super-classes of *Abs* are linked by equivalence modulo β. Although superficially different, the notion of local coupling is closely related to admissible invariant.

Table 7. The induced coupling relation for Def. 13

$\mathscr{R}\,\beta\,\theta\,\alpha\,\alpha'$	$\Leftrightarrow \alpha \sim_\beta \alpha'$ if θ is **bool**, C, Γ, or $\mathsf{state}\,C$
$\mathscr{R}\,\beta\,(\mathsf{heap}\otimes\Gamma)\,(h,s)\,(h',s')$	$\Leftrightarrow \mathscr{R}\,\beta\,\mathsf{heap}\,h\,h' \wedge \mathscr{R}\,\beta\,\Gamma\,s\,s' \wedge \mathit{disciplined}(h,s) \wedge \mathit{disciplined}(h',s')$
$\mathscr{R}\,\beta\,(\mathsf{heap}\otimes T)\,(h,v)\,(h',v')$	$\Leftrightarrow \mathscr{R}\,\beta\,\mathsf{heap}\,h\,h' \wedge \mathscr{R}\,\beta\,T\,v\,v'$
$\mathscr{R}\,\beta\,(\theta_\bot)\,\alpha\,\alpha'$	$\Leftrightarrow (\alpha = \bot = \alpha') \vee (\alpha \neq \bot \neq \alpha' \wedge \mathscr{R}\,\beta\,\theta\,\alpha\,\alpha')$
$\mathscr{R}\,\beta\,(\Gamma \vdash T)\,v\,v'$	$\Leftrightarrow \forall h,s,h',s' \mid \mathscr{R}\,\beta\,(\mathsf{heap}\otimes\Gamma)\,(h,s)\,(h',s')$ $\Rightarrow \mathscr{R}\,\beta\,T_\bot\,(v(h,s))\,(v'(h',s'))$
$\mathscr{R}\,\beta\,(C,\bar{x},\bar{T}\to T_1)\,v\,v'$	$\Leftrightarrow \forall h,s,h',s' \mid \mathscr{R}\,\beta\,(\mathsf{heap}\otimes\Gamma)\,(h,s)\,(h',s')$ $\Rightarrow \exists \beta_0 \supseteq \beta \mid \mathscr{R}\,\beta_0\,(\mathsf{heap}\otimes T_1)_\bot\,(v(h,s))\,(v'(h',s'))$ where $\Gamma = [\bar{x}:\bar{T}, \mathsf{self}:C]$
$\mathscr{R}\,\mathsf{menv}\,\mu\,\mu'$	$\Leftrightarrow \forall C,m,\beta \mid \mathscr{R}\,\beta\,(C,\bar{x},\bar{T}\to T)\,(\mu C m)\,(\mu' C m)$ where $\mathit{mtype}(m,C) = \bar{T}\to T$ and $\mathit{pars}(m,C) = \bar{x}$

In applications, $\mathscr{L}\,\beta\,h\,h'$ would be defined as something like this: h and h' partition as islands $Ah * Rh * Sh$ and $Ah' * Rh' * Sh'$ such that $Ah * Rh * Sh \models \mathscr{I}^{Abs}$ and $Ah' * Rh' * Sh' \models \mathscr{I}'^{Abs}$ and some condition links the data structures [18]. The bijection β would not be explicit but would be induced as a property of the formula language.

A local coupling $\mathscr{L}$ induces a relation on arbitrary heaps by requiring that they partition such that islands can be put in correspondence so that pairs are related by $\mathscr{L}$.

Definition 13 (coupling relation, $\mathscr{R}$). Given local coupling $\mathscr{L}$, we define for each θ and β a relation $\mathscr{R}\,\beta\,\theta \subseteq [\![\theta]\!] \times [\![\theta]\!]'$ as follows.

For heaps h,h', we define $\mathscr{R}\,\beta\,\mathsf{heap}\,h\,h'$ iff h,h' are disciplined, $\beta \subseteq \mathit{dom}\,h \times \mathit{dom}\,h'$, and β fully partitions h,h' for Abs; moreover, if the Abs-partitions are $h = Fh * Ah_1 * Rh_1 * Sh_1 \ldots Ah_k * Rh_k * Sh_k$ and $h' = Fh' * Ah'_1 * Rh'_1 * Sh'_1 \ldots Ah'_k * Rh'_k * Sh'_k$ then (recall Lemma 4) (a) β restricts to a total bijection between $\mathit{dom}(Fh)$ and $\mathit{dom}(Fh')$; (b) $Fh \sim_\beta Fh'$; and (c) for all i,j, if $\beta\,(\mathit{pickdom}\,Ah_i)\,(\mathit{pickdom}\,Ah'_j)$ then

- β restricts to a total bijection between $\mathit{dom}(Sh_i)$ and $\mathit{dom}(Sh'_j)$
- $(Ah_i * Sh_i) \sim_\beta (Ah'_j * Sh'_j)$
- $h(\mathit{pickdom}\,Ah_i).\mathit{inv} \leq Abs \Rightarrow \mathscr{L}\,\beta\,(Ah_i * Rh_i * Sh_i)\,(Ah'_j * Rh'_j * Sh'_j)$

For other categories θ we define $\mathscr{R}\,\beta\,\theta$ in Table 7.

The third item under (c) is the key connection with the $\mathit{inv}/\mathit{own}$ discipline.

Under the antecedent in the definition, $(Ah_i * Sh_i) \sim_\beta (Ah'_j * Sh'_j)$ is equivalent to the conjunction of $Ah_i \sim_\beta Ah'_j$ and $Sh_i \sim_\beta Sh'_j$. And $Ah_i \sim_\beta Ah'_j$ means that the two objects o,o' agree on superclass and subclass fields (but not the declared fields of Abs); in particular, $\mathit{type}\,o = \mathit{type}\,o' \leq Abs$ and $Ah_i\,o.\mathit{inv} = Ah'_j\,o'.\mathit{inv}$.

The gist of the abstraction theorem is that if methods of Abs are related by $\mathscr{R}$ then all methods are. In terms of the preceding definitions, we can express quite succinctly the conclusion that all methods are related: $\mathscr{R}\,\mathsf{menv}\,[\![CT]\!]\,[\![CT']\!]'$. We want the antecedent of the theorem to be that the meaning $[\![M]\!]$ is related to $[\![M']\!]'$, for any m with declaration M in $CT(Abs)$ and M' in $CT'(Abs)$. Moreover, $[\![M]\!]$ depends on a method environment. Thus the antecedent of the theorem is that $[\![M]\!]\mu$ is related to $[\![M']\!]'\mu'$ for all related μ,μ'. (It suffices for μ,μ' to be in the approximation chains defining $[\![CT]\!]$ and $[\![CT']\!]'$).

5.3 Simulation and the Abstraction Theorem

Definition 14 (simulation). A simulation is a coupling $\mathscr{R}$ such that the following hold.

- ($\mathscr{L}$ is initialized) For any $C \leq Abs$, and any o, o' with $\beta\, o\, o'$ and $type\, o = C$ we have $\mathscr{L}\, \beta\, h\, h'$ where $h = [o \mapsto [dom(xfields\, C) \mapsto defaults\, C]]$ and $h' = [o' \mapsto [dom(xfields'\, C) \mapsto defaults'\, C]]$.
- (methods of Abs preserve $\mathscr{R}$) For any disciplined μ, μ' such that $\mathscr{R}\ \mathsf{menv}\ \mu\ \mu'$ we have the following for every m declared in Abs. Let $\bar{U} \to U = mtype(m, Abs)$ and $\bar{x} = pars(m, Abs)$. For every β, we have $\mathscr{R}\ \beta\ \theta\ ([\![M]\!]\mu)\ ([\![M']\!]'\mu')$ where $\theta = (Abs, \bar{x}, \bar{U} \to U)$. where M (resp. M') are as above. (We omit the similar condition for inherited methods.)

Lemma 5 (preservation by expressions). For all expressions $\Gamma \vdash e : T$ that contain no field access subexpressions, and all β, we have $\mathscr{R}\, \beta\, (\Gamma \vdash T)\ ([\![\Gamma \vdash e : T]\!])\ ([\![\Gamma \vdash e : T]\!]')$.

Lemma 6 (preservation by commands). Let μ, μ' be disciplined method environments with $\mathscr{R}\ \mathsf{menv}\ \mu\ \mu'$. If $\Gamma \vdash S$ is a properly annotated command for Abs, with $\Gamma\, \mathsf{self} \neq Abs$, then for all β we have the following. If $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$ and $\neg(\exists o \mid o \succ^{h}_{Abs} s(\mathsf{self}))$ and $\neg(\exists o' \mid o' \succ^{h'}_{Abs} s'(\mathsf{self}))$ then there is $\beta_0 \supseteq \beta$ such that $\mathscr{R}\ \beta_0\ (\mathsf{heap} \otimes \Gamma)_{\bot}\ (v(h, s))\ (v'(h', s'))$.

Our main result says that if methods of Abs preserve the coupling then all methods do.

Theorem 1 (abstraction).
If $\mathscr{R}$ is a simulation for comparable class tables CT, CT' then $\mathscr{R}\ \mathsf{menv}\ [\![CT]\!]\ [\![CT']\!]'$.

6 Using the Theorem

A complete program is a command S in the context of a class table. To show equivalence between CT, S and CT', S, one proves simulation for Abs and then appeals to the abstraction theorem to conclude that $[\![S]\!]$ is related to $[\![S]\!]'$. Finally, one appeals to an *identity extension lemma* that says the relation is the identity for programs where the encapsulated representation is not visible. We choose simple formulations that can also serve to justify more specification-oriented formulations. We say that a state (h, s) is *Abs-free* if $type\, o \not\leq Abs$ for all $o \in dom\, h$.

Lemma 7 (identity extension). If $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$
then $encapAbs\, (h, s) \sim_{\beta} encapAbs\, (h', s')$.

Lemma 8 (inverse identity extension). Suppose (h, s) and (h', s') are Abs-free. If $(h, s) \sim_{\beta} (h', s')$ and β is total on h, h' then $\mathscr{R}\ \beta\ (\mathsf{heap} \otimes \Gamma)\ (h, s)\ (h', s')$.

Definition 15 (program equivalence). Suppose programs $CT, (\Gamma \vdash S)$ and $CT', (\Gamma \vdash' S')$ are such that CT, CT' are comparable and properly annotated, and moreover S, S' are properly annotated. The programs are *equivalent* iff for all disciplined, *Abs*-free (h,s) and (h',s') in $[\![\mathsf{heap} \otimes \Gamma]\!]$ and all β with β total on h,h' and $(h,s) \sim_\beta (h',s')$, there is some $\beta_0 \supseteq \beta$ with $encapAbs([\![\Gamma \vdash S]\!]\mu(h,s)) \sim_{\beta_0} encapAbs([\![\Gamma \vdash' S']\!]'\mu'(h',s'))$ where $\mu = [\![CT]\!]$ and $\mu' = [\![CT']\!]'$.

Proposition 3 (simulation and equivalence). Suppose programs $CT, (\Gamma \vdash S)$ and $CT', (\Gamma \vdash' S)$ are properly annotated and $\mathscr{R}$ is a simulation from CT to CT'. If $\Gamma\,\mathsf{self} \neq Abs$ then the programs are equivalent.

7 Discussion

Adaptations of the inv/own Discipline. As compared with previous work on the discipline, we have imposed some additional restrictions to achieve sufficient information hiding to justify a modular rule for equivalence of class implementations. We argue that the restrictions are not onerous for practical application, though further practical experience is needed with the discipline and with our rule.

The first restriction is on field reads. Code in a client class cannot be allowed to read a field of an encapsulated representation object, although the discipline allows the existence of the reference; otherwise the client code could be representation dependent. On the other hand, a class such as *Hashtable* might be used both by clients and in the internal representation of the class *Abs* under revision; certainly the code of *Hashtable* needs to read its own fields. A distinction can be made on the basis of whether the current target object, i.e., self, is owned by an instance *o* of *Abs*. If it is, then we do not need the method invocation to preserve the coupling and we can allow reading of objects owned by *o*. If the target object is not owned by an instance of *Abs* then it should have no need to access objects owned by *Abs*. This distinction appears in the statement of Lemma 6 and it is used to stipulate a precondition for field access (see Def. 5).[4]

Because the coupling relation imposes the user-defined local coupling only when an *Abs*-object is packed, it appears necessary to restrict **pack** *e* **as** *Abs* to occur only in code of *Abs* in order for simulation to be checked only for that code. In the majority of known examples, packing to a class *C* is only done in code of *C*, and this is required in Leino and Müller's extension of the discipline to handle static fields.

Similar considerations apply to **setown** *o* **to** (p,C): care must be taken to prevent arbitrary code from moving objects across the encapsulation boundary for *Abs* in ways that do not admit modular reasoning. One would expect that code outside *Abs* cannot move objects across the *Abs*-boundary at all, but it turns out that the only problematic case is transfer out from an *Abs* island. In the unusual case that **setown** *o* **to** (p,C) occurs in code outside *Abs* but *o* is initially inside the island for some *Abs*-object, then

[4] This is unattractive in that the other stipulated preconditions mention only direct ownership whereas this one uses transitive ownership. But in practical examples, code outside *Abs* rarely has references to encapsulated objects. We believe such references can be adequately restricted using visibility control and/or lightweight confinement analyses, e.g., [31, 2].

o must end up in the island for some *Abs*-object. Our stipulated precondition says just this. In practice it seems that the obligation can be discharged by simple syntactic considerations of visibility and/or lightweight alias control.

The last restriction is that an *Abs* object cannot own other *Abs* objects. This does not preclude containers holding containers, because a container does not own its content (e.g., AQueue owns the Qnodes but not the tasks). It does preclude certain recursive situations. For example, we could allow Qnode instances to own their successors but then we could not instantiate the theory with *Abs*:=Qnode. This does not seem too important since it is Queue that is appropriate to view as an abstraction coupled by a simulation. The restriction is not needed for soundness of simulation. But absent the restriction, nested islands would require a healthiness condition on couplings (similar to the healthiness condition used by Cavalcanti and Naumann [13–Def. 5]); e.g., coupling for an instance of Qnode would need to recursively impose the same predicate on the nxt node. We disallow nested islands in the present work for simplicity and to highlight connections with separation logic.

Future Work. The discipline may seem somewhat onerous in that it uses verification conditions rather than lighter weight static analysis for control of the use of aliases. (We have to say "use of", because whereas confinement disallows certain aliases, the invariant discipline merely prevents faulty exploitation of aliases.) The Spec# project [7] is exploring the inference of annotations. For many situations, simple confinement rules and other checks are sufficient to discharge the proof obligations and this needs to be investigated for the additional obligations we have introduced. The advantage of a verification discipline over types is that, while simple cases can be checked automatically, complicated cases can be checked with additional annotations rather than simply rejected.

The generalization to a small group of related classes is important, as revisions often involve several related classes. One sort of example would be a revision of our Queue example that involves revising Qnode as well. If nodes are used only by Queue then this is subsumed by our theory, as we can consider a renamed version of Qnode that coexists with it. The more interesting situations arise in refactoring and in design patterns with tightly related configurations of multiple objects. The friend and peer dependencies of [21, 9, 28], and the flexible ownership system of Aldrich and Chambers [1] could be the basis for a generalization of our results.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.
2. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, revision pending. Extended version of [3].
3. A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *POPL*, 2002.
4. A. Banerjee and D. A. Naumann. Ownership transfer and abstraction. Technical Report TR 2004-1, Computing and Information Sciences, Kansas State University, 2003.

5. A. Banerjee and D. A. Naumann. State based encapsulation and generics. Technical Report CS Report 2004-11, Stevens Institute of Technology, 2004.
6. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3, 2004.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS post-proceedings*, 2004.
8. M. Barnett, D. A. Naumann, W. Schulte, and Qi Sun. 99.44% pure: useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs*, 2004.
9. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction*, 2004.
10. P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A refinement algebra for object-oriented programming. In *ECOOP*, 2003.
11. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
12. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, 2001.
13. A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In *Formal Methods Europe*, 2002.
14. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, 2002.
15. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
16. D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research 156, DEC Systems Research Center, 1998.
17. J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
18. C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1, 1972.
19. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Syst.*, 23, 2001.
20. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *International Symposium on Software Security*, 2003.
21. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.
22. B. Meyer. *Object-oriented Software Construction*. Second edition, 1997.
23. I. Mijajlovic, N. Torp-Smith, and P. O'Hearn. Refinement and separation contexts. In *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, 2004.
24. J. C. Mitchell. Representation independence and data abstraction. In *POPL*, 1986.
25. P. Müller, A. Poetzsch-Heffter, and G. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zürich, Oct. 2003.
26. D. A. Naumann. Observational purity and encapsulation. In *FASE*, 2005.
27. D. A. Naumann. Verifying a secure information flow analyzer. To appear in *TPHOLS*, 2005.
28. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *LICS*, 2004.
29. P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, 2004.
30. F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP*, 2000.
31. J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31, 2001.