Under consideration for publication in Formal Aspects of Computing

On assertion-based encapsulation for object invariants and simulations

David A. Naumann[1]

Department of Computer Science, Stevens Institute of Technology

Abstract. In object-oriented programming, reentrant method invocations and shared references make it difficult to achieve adequate encapsulation for sound modular reasoning. This tutorial paper surveys recent progress using auxiliary state (ghost fields) to describe and achieve encapsulation. It also compares this technique with encapsulation in the forms provided by separation logic. Encapsulation is assessed in terms of modular reasoning about invariants and simulations.

Keywords: Object invariants; Encapsulation and abstraction; Separation and alias control.

1. Introduction

Reentrant callbacks and sharing of mutable objects are ubiquitous in object-oriented programs and both cause problems in reasoning. This paper presents an approach to modular reasoning based on the addition of ghost ("fictitious", auxiliary) fields with which intended structural relationships can be expressed —in particular, dependency relationships. The approach facilitates reasoning about object invariants and simulation relations. Object invariants are essential for modular proof of correctness and simulations are essential for modular proof of equivalence or refinement of class implementations. The approach offers interdependent solutions to the two problems. It was developed initially by Barnett et al. [BDF+04] and has been extended by Leino, Müller, and others [LM04, BN04, NB06, PCdB05].

Besides giving a tutorial introduction to the approach, we compare it with other approaches and suggest possible extensions and opportunities for future work. Müller et al. [MPHL04] and Jacobs et al. [JKW03] give good introductions to the problems and other solution approaches. The book by Szyperski et al. [SGM02] illustrates the reentrant callback problem using more realistic examples than can fit in a research paper. We assume the reader has minimal familiarity with Java-like languages; some familiarity with design patterns [GHJV95] may be helpful. The problems addressed are related to issues in frame specifications, but that is not the focus here. Nor do we address concurrency, though there are important connections (see, e.g.,[Jon96, JLS04]).

Correspondence and offprint requests to: David A. Naumann, Department of Computer Science, Stevens Institute of Technology, Castle Point on Hudson, Hoboken NJ 07030 USA. e-mail: naumann@cs.stevens.edu

[1] Supported in part by the US National Science Foundation, under grants CCR-0208984 and CCF-0429894, and by Microsoft Research.

Outline. Section 2 sketches the problems. Section 3 addresses invariants and reentrant callbacks in detail and how they are handled in the ghost variable approach. Section 4 addresses invariants and object sharing using a notion of ownership. Section 5 discusses additional issues concerning ownership-based invariants and Section 6 shows how the approach can be used for simulations. Section 7 considers an extension of the approach to invariants that depend on non-owned objects —a sort of rely-guarantee discipline for friendly cooperation. Section 8 revisits the approach from the perspectives of separation logic and another recent proposal. Section 9 discusses prospects and challenges for further extensions.

2. How shared objects and reentrant callbacks violate encapsulation

Several constructs in Java and similar programming languages are intended to provide encapsulation. A *package* collects interrelated classes and serves as a unit of scope. Each instance of a *class* provides some abstraction, as simple as a complex number or as complex as a database server. A *method specification* describes an operation in terms of the abstraction.[2] A method *implementation* uses other abstractions and is verified, for the sake of modularity, with respect to their specifications.

Less frequently, a class itself provides some abstraction, represented using static fields.[3] More frequently, instances of multiple classes collectively provide an abstraction of interest, e.g., a collection and its iterators. In this paper we focus on the case of an abstraction provided by a single instance or small group of instances.

To show that a method implementation satisfies its specification it is often essential to reason in terms of an *object invariant*[4] for the target or receiver object $\mathbf{self}$. An object's invariant involves consistency conditions on internal data structures —its representation, made up of so-called *rep objects*— and the connection with the abstraction they represent. To a first approximation, an object's invariant is an implicit precondition and postcondition for every method of its class, as described in Hoare's early work on abstract data types [Hoa72]. The invariant should not be explicit in a method's public specification, since it involves the representation. To maintain the invariant, it should suffice for the methods of the class to maintain it since, owing to encapsulation, it is not susceptible to interference by client code.

These notions are clear and effective in situations where abstractions are composed by hierarchical layering. However, both reentrant callbacks and object sharing can violate simple hierarchical structure.

Reentrant callbacks. Consider some kind of sensor playing the role of Subject in the Observer design pattern [GHJV95]. The sensor maintains a set of registered Views: when the sensor value reaches the threshhold, $v.thresh$, of a given view v, the sensor invokes method $v.notify()$ and removes v from the set. This description is in terms of a set, part of the abstraction offered by the Subject; the implementation might store views in an array ordered by $thresh$ values. The pattern cannot be seen simply as a client layered upon an abstraction: $notify$ is an *upcall*, to the client. The difficulty is that $v.notify$ may make a *reentrant callback* to the sensors. Consider the following sequence of invocations, where s is a sensor: A client or asynchronous event invokes $s.update()$ which changes the value of the sensor. Before returning, $update$ invokes $v.notify()$ where v is a view registered with s. Now v maintains a reference, $v.sensor$, to s and in order for $notify$ to do its job it invokes $v.sensor.getval()$ to determine the current sensor value. Because $v.sensor = s$, this invocation of $getval$ is known as a *reentrant callback*: control returns to s while another method invocation (here $update$) is already in progress.

It is common that reentrant callbacks are intended. In the example, $getval$ might simply read a field and cause no problem. However, trouble is likely if v invokes on s a method $enum$ that enumerates the current set of views of s, since $enum$ likely depends on the invariant that the array of views is in a consistent state —is v still in the set of registered views? In terms of the array, is v in fact in the array?

According to our first approximation, the invariant is required as a precondition for method $update$ and must be reestablished by it, but need not hold at the intermediate point where $s.update$ invokes $v.notify$. Nor, according to the first approximation, should $notify$ be responsible for establishing the invariant. The first approximation needs to be refined in regard to both when invariants hold and which code is responsible.

---

[2] The term "method" is used in the sense of object-oriented programming, i.e., of procedures dispatched on the basis of the target object's allocated type.

[3] Associated with a class rather than with each instance.

[4] In a class-based language it is natural to include in a class the declaration of an invariant, with the interpretation that each instance satisfies the invariant. To emphasize the instance-oriented nature of such invariants, we use the term *object invariant* although some authors prefer "class invariant".

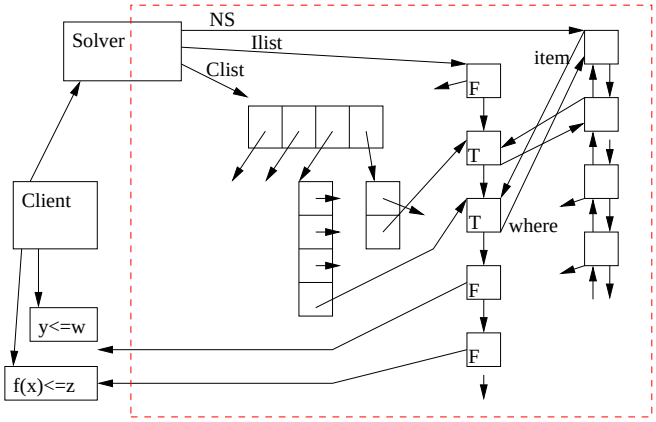


Fig. 1. Some of the objects in a data structure. Labels indicate names of some pointer fields.

This example involves cyclic linking $s \rightarrow v \rightarrow s$ of heap objects. We consider next a different problem due to shared references.

Shared mutable objects. An illustration of the challenging invariants found in object-oriented programs is the structure of objects and references in Fig. 1. This depicts the data structures used in a constraint solving algorithm [RM99]. Several structural invariants must be maintained by $Solver$ for correctness and efficiency of the algorithm. For example, the objects in the vertical column on the far right form a doubly linked list rooted at NS and their $item$ fields point to elements of the list rooted at $Ilist$. Moreover, each of those items is in the range of the array of arrays $Clist$. And there is cross-linking between $Ilist$ and NS. These examples can be written as follows:[5]

$$
\begin{array}{l}
(NS = \mathbf{null} \vee NS.prev = \mathbf{null}) \\
\wedge\, (\forall\, p \in NS.next^* \mid (p.next = \mathbf{null} \vee p.next.prev = p) \\
\qquad\qquad \wedge\, p.item \in Ilist.next^* \wedge (\exists\, x, j \mid Clist[x][j] = p.item\ ) \ \wedge\ p.item.where = p\ )
\end{array}
$$

The client program is intended to have a reference to the $Solver$ object. The data structure includes pointers to client objects that represent constraints (e.g., "$y \leq w$"). The latter pointers go against a strict hierarchical layering of abstractions (clients using solvers), but this is not necessarily a problem. But there is no reason for clients to have references to the objects within the dashed boundary; these are intended to comprise the encapsulated representation of the solver. A reference to one of these rep objects would be problematic because the client could update the object and falsify the invariant of $Solver$, contrary to the expectation of encapsulation.

Suppose for simplicity that class $Solver$ has no proper subclasses, nor superclasses other than $Object$ (which is reasonable in this example since the class basically provides a single algorithm). Fields NS, $Ilist$, and $Clist$ can be given private scope so no code of other classes can update them. We can reason in a modular way about invariants that depend only on these fields, e.g., $Clist \neq \mathbf{null}$. If such an invariant is established by the constructor then —absent reentrant callbacks— we can assume it as a precondition of every method of $Solver$ so long as it is established as a postcondition of every method of $Solver$.

The formula displayed above, however, depends on fields of other objects besides the $Solver$; scope-based encapsulation does not protect them from interference by client code. For example, if the client held a reference o to the first node in NS, i.e., $o = NS$ with $NS \neq \mathbf{null}$, then it could set $o.prev := o$, violating the first line of the invariant which enforces acyclicity. If a method of $Solver$ is then invoked, it is not sound to assume the invariant as a precondition.

Scope alone is inadequate, and it would seem that heap encapsulation needs to be expressed somehow in terms of state. A notion of heap encapsulation that fits this situation is *ownership*. The idea has three parts. First, the objects comprising the representation of an instance of $Solver$ are considered to be owned by it —the encapsulation boundary encloses exactly the owned objects. Second, the invariant is only allowed to depend on owned objects. Third, invariant-falsifying updates are prevented by some means. The most common means is to disallow references to owned objects

[5] Here $*$ denotes reflexive transitive closure of field dereferences. We use a Java-like notation with implicit dereferencing: $p.next$ is the value of field $next$ in the object pointed to by p.

```
class Subject{
  private x, y : int := 0, 1;   view : View := ...;
  invariant I^Subject   where I^Subject =df (self.x < self.y)
  method m() { x := x + 1; view.notify(); y := y + 1; }
  method f() : float { return 1/(y − x); }
}
class View {
  z : Subject := ...;
  method notify() { ... z.f(); }
}
```

Fig. 2. Simple example of reentrant callback. (Occurrence of a field name like x without qualifier abbreviates $\mathbf{self}.x$.)

from outsiders. This is the *dominator property* [Cla01]: Every path to a rep of $Solver\ s$ from an object that is not a rep of s must go through s. Ownership is the topic of Section 4.

3. Reentrance and object invariants

In this section we set aside ownership and present a discipline for invariants in the presence of reentrant callbacks.

For clarity we use the contrived example in Fig. 2. In class $Subject$, the object invariant $x < y$ is established by initialization $x, y := 0, 1$. Method f relies on the invariant (to avoid division by 0); it maintains the invariant because it does no updates. At first glance, the invariant is also maintained by m since it increments x and y by the same amount. Because x and y are local, they are not susceptible to update in code outside of class $Subject$ —and this is what we need for modular reasoning about $Subject$. But there is the possibility of a reentrant callback. For object s of type $Subject$, an invocation of $s.m$ results in the invocation $s.view.notify$ in a state where the declared invariant does not hold for s. Now $s.view.notify$ in turn invokes $z.f$, so if $s.view.z = s$ then an error occurs. If instead $notify$ invoked $z.m$ then the program would diverge due to nonterminating recursion.

Possible solutions. One reaction to the example is to disallow any reentrant callbacks. This could be done using static analysis for control flow. It is not straightforward, because cycles in a pure calling graph are very common, e.g., owing to recursion over tree-like data structures as well as various design patterns. To avoid rejecting too many programs, the analysis needs to take aliasing into account. Such analyses are usually not modular, however. A specification of allowed calling patterns might also be required, since for example if f simply returned x then the callback $m \rightarrow notify \rightarrow f$ is harmless and possibly desirable.

The problem is similar to interference found in concurrent programs and one might try to solve it using locks. But here we are concerned with a single thread of control; if a lock was taken by the initial call to m and that lock prevented a reentrant call then deadlock would result. (In Java, a lock held by a given thread does *not* prevent that thread from reentering the object, precisely to avoid deadlock.) A related solution is to introduce a boolean field inm to represent that a call of m is in progress and to use $\neg inm$ as precondition of m and f. This has similarities to the approach advocated later in this paper.

Another approach to the problem is to require the invariant to hold prior to any method call, lest that call lead to a reentrant callback. This has been advocated in the literature [LG86, Mey97] and is sometimes called *visible state semantics* [MPHL04]. Our example can be revised to fit this discipline, by changing the body of m to this:

$$x := x + 1;\ y := y + 1;\ view.notify(); \tag{1}$$

Note that $\mathcal{I}^{Subject}$ holds after the second assignment so it is sound for $view.notify$ to rely on it, e.g., by making reentrant calls. But this approach does not scale to more complicated programs, where the invariant may involve several data structures, update of which is done by method calls. Most method calls do not lead to reentrant callbacks and we already noted that some reentrant callbacks are harmless, even desirable.

Another alternative would be to state the invariant as an explicit precondition for those methods that depend on it. Then $notify$ in the example would be rejected because it could not establish the precondition for its call $z.f()$. This alternative must be rejected on grounds of information hiding: the predicate $\mathcal{I}^{Subject}$ depends on internals that should be encapsulated within class $Subject$ and not visible to $View$.

Various techniques have been proposed to hide information about an invariant while expressing in the specification whether it is in force. One alternative is to introduce a typestate [DF01] to stand for “the invariant is in force”. Another

```
class Subject {
  private x, y : int := 0, 1;   view : View := ...;
  invariant I^Subject   where I^Subject =_df self.x < self.y
  method m()
    requires self.inv
    ensures self.inv
  { unpack self; assert ¬self.inv; x := x + 1; view.notify(self); y := y + 1; }
}
class View {
  method notify(Subject z) {z.m();}
}
```

Fig. 3. Variation on Fig. 2 (incomplete).

approach is to treat its name as opaque with respect to its definition [BP05]; this is pursued in Section 8. Another way to treat the invariant as an opaque predicate, which to the author's knowledge has not been explored, is to use a pure method [LCC+03] to represent the invariant. This could be of practical use in runtime verification and hiding of internals could be achieved using visibility rules of the programming language.

The Boogie approach. The best features of the preceding alternatives are combined in the so-called Boogie methodology Barnett et al. [BDF+04]. The idea is to make explicit in preconditions not the invariant predicate, e.g., $\mathcal{I}^{Subject}$, but rather a boolean abstraction of it (similar to the typestate approach). For reasons that will become clear, we use the term *inv/own discipline* for the Boogie approach.

To a first approximation, the discipline uses a *ghost* (auxiliary) field inv of type boolean so that $o.inv$ represents the condition that "the invariant $\mathcal{I}[o/\mathbf{self}]$ is in force". The idea is that the implication $o.inv \Rightarrow \mathcal{I}[o/\mathbf{self}]$ can be made to hold in every state, while $\mathcal{I}[o/\mathbf{self}]$ itself is violated from time to time for field updates. The idea can be used with an ordinary field, but here we use a ghost field that has no runtime significance but rather is used only for reasoning. Field inv is considered to be public and declared in the root class $Object$, so inv can appear as a precondition of any method that depends on the invariant. For our running example, both methods f and m would have precondition $\mathbf{self}.inv$.

The discipline imposes several proof obligations in order to ensure that the following is a *program invariant* (i.e., it holds in any state reachable in any computation):

$$(\forall\, o \mid\ o.inv \Rightarrow \mathcal{I}[o/\mathbf{self}]\ ) \tag{2}$$

(We let o range over references to all allocated object references.) Consider a method m with (at least) precondition $\mathbf{self}.inv$. To reason about correctness of an implementation of m, within the scope where $\mathcal{I}$ is visible, the conjunction of (2) and $\mathbf{self}.inv$ yield $\mathcal{I}$ as a precondition. On the other hand, outside the scope a reasoner sees only the precondition $\mathbf{self}.inv$.

To emphasize that inv is a ghost variable used only for reasoning, the discipline uses special commands **pack** and **unpack** to set inv true and false, respectively. Key proof obligations are imposed on these. The obligations are most easily understood in terms of allowed proof outlines. In particular, certain preconditions are stipulated for the special commands and for field updates. The two most important obligations are as follows:

- The precondition for **pack** E is $\neg E.inv \wedge \mathcal{I}[E/\mathbf{self}]$. Clearly $\mathcal{I}[E/\mathbf{self}]$ is necessary to maintain (2), since **pack** sets $E.inv$ to true. The first conjunct prevents reentrance to this region of code.
- The precondition for a field update $E.f := E'$ is $\neg E.inv$. This ensures that the update does not falsify (2) for the object E.

Consider the code in Fig. 3, a variation on Fig. 2. Here the $Subject$ passes **self** as an argument to $notify$ and an incomplete annotation is sketched. The update $x := x + 1$, which abbreviates $\mathbf{self}.x := \mathbf{self}.x + 1$, is subject to precondition $\neg\mathbf{self}.inv$. As the precondition of m is $\mathbf{self}.inv$, the special command **unpack self** is needed to set inv false. Now we consider some options in reasoning about $notify$.

One possibility is for $z.inv$ to be a precondition for $notify$. Then the implementation of $notify$ is correct: according to the specification of m, the call $z.m()$ has precondition $z.inv$. The implementation of $Subject.m$ is thus forced to establish $\mathbf{self}.inv$ preceding the call to $notify$.

Setting $\mathbf{self}.inv$ true is the effect of the special command **pack self**, but the stipulated precondition for **pack self** is $\mathcal{I}$ and this assertion would not hold immediately following the update $x := x + 1$. The situation can be repaired as

```
class Integer {
  public val : int;
  method incr() { val := val + 1; }
}
class Subject2 {
  private x : Integer := new Integer(0);
  private y : Integer := new Integer(1);
  invariant I^Subject2   where I^Subject2 =_df (x ≠ null ≠ y ∧ x.val < y.val)
  method m()
    requires self.inv
    ensures self.inv
  { unpack self;
    assert I^Subject2; x.incr(); y.incr(); assert I^Subject2;
    pack self; }
  method leak() : Integer { return x; }
}
class Main{   s : Subject2 := new Subject2; ...
  method main() { i : Integer := s.leak(); i.incr(); s.m(); }
}
```

Fig. 4. Invariant dependent on rep objects.

in the following possible implementation of m, where, as in (1), the assignment $y := y + 1$ precedes invocation of $notify$ so that the implementation of m can be verified.

$$\mathbf{unpack\ self}; \mathbf{assert}\ \neg\mathbf{self}.inv; x := x + 1; y := y + 1; \mathbf{assert}\ \mathcal{I}^{Subject}; \mathbf{pack\ self}; view.notify(\mathbf{self});$$

This seems satisfactory for the example but in general it is impractical to impose the visible state semantics for invariants. The example does show that the discipline can handle this pattern of reasoning.

Another possibility is to retain the implementation of m, so that inv is only restored at the end, as in the following:

$$\mathbf{unpack\ self};\ \mathbf{assert}\ \neg\mathbf{self}.inv;\ x := x + 1;\ view.notify(\mathbf{self});\ y := y + 1;\ \mathbf{assert}\ \mathcal{I}^{Subject};\ \mathbf{pack\ self};$$

For the call to $notify$ to be correct, $notify$ cannot have precondition $z.inv$. But then the implementation of $notify$ is not correct, because it has no way to establish $z.inv$ which is the precondition for $z.m$. On the other hand, $notify$ is free to invoke on z any method that does not require $z.inv$. And this is the most typical scenario for this pattern.

In summary, the discipline uses ghost field inv in such a way that harmful reentrant callbacks can be prevented while allowing some callbacks. There is a clear intuition, that $z.inv$ stands for "z is in a consistent state" (it is *packed*, for short). Yet the internal representation of $Subject$ is not exposed to $View$; there is no need for predicate $\mathcal{I}^{Subject}$ to be visible outside $Subject$. Invariants of the form $inv \Rightarrow \mathcal{I}$ are made to hold in every state, in particular as precondition for every method call.

4. Sharing and object invariants

Let us set aside the issue of reentrance and turn attention to shared references, while retaining the inv idea. Consider the toy example in Fig. 4. The initialization of $Subject2$ establishes $\mathcal{I}^{Subject2}$. The annotation of m is correct: The first assertion follows from precondition $\mathbf{self}.inv$ and program invariant (2). The second assertion follows from the first by straightforward reasoning about $incr$. Method $leak$ does no updates and thus maintains $\mathcal{I}^{Subject2}$.

Unfortunately, $main$ uses $leak$ to falsify (2). In a state where $s.inv$ is true, and thus $\mathcal{I}^{Subject2}[s/\mathbf{self}]$ holds by (2), $main$ uses $s.leak()$ to obtain a (shared) reference i to $s.x$. The invocation $i.incr()$ then updates the val field, falsifying $s.x.val < s.y.val$ and thus falsifying $s.inv \Rightarrow \mathcal{I}^{Subject2}[s/\mathbf{self}]$.

One diagnosis is that the invariant of $Subject2$ should not be allowed to depend on fields of objects other than $\mathbf{self}$. Indeed, some proposals in the literature on invariants for object-oriented programs are only sound under this restriction [LW94]. But for many classes this is highly impractical, for example, the $Solver$ in Section 2.

The name "$leak$" indicates our diagnosis: just as field x is private, so too the object referenced by x belongs within class $Subject2$ in some sense. More precisely, it is a *rep* object —part of the representation of an abstraction provided

by an instance of $Subject2$. A rep belongs to its owner and this licenses its owner's invariant to depend on it. Thus the programming discipline must prevent updates of reps by code outside $Subject2$.

Ownership. As mentioned in Section 2, some ownership systems prevent harmful updates by preventing the existence of references from client to rep (the dominator property that all paths to a rep go through its owner). It is easy to violate the dominator property: a method could return a rep pointer, pass it as an argument to a client method, or store it in a global variable.

The dominator property can be enforced using a type system such as the Universe system [Mül02] and variations on Ownership Types [CNP01, BLS03, BLR02, AC04]. These systems do not directly enforce the dominator property, which is expressed in terms of paths. Rather, they constrain references, disallowing any object outside an ownership domain from having a pointer to inside the domain. This means that from the point of view of a particular object s, the heap can be partitioned into three blocks:

- the singleton containing just s
- the objects owned by s (which, together with s, are called an *island*)
- all other objects

In these terms, the invariant for s is only allowed to depend on fields of objects in the island of s. The dashed box in Fig. 1 depicts an island. (The idea and the term are both due to Hogg [Hog91].)

The name "$leak$" suggests that what has gone wrong in the example of Fig. 4 is the very existence of a shared reference. Ownership type systems prevent harmful updates by alias control: static rules would designate that x is owned and would reject method $leak$. This approach has attractive features but it has proved difficult to find an ownership type system that admits common design patterns and also enforces sufficiently strong encapsulation for modular reasoning about object invariants. In particular, many examples call for the transfer of ownership (see Section 5) which does not sit well with type-based systems. Moreover ownership typing involves rather special program annotations (decorating declarations with ownership information).

The alternative presented below controls *uses* of references and represents ownership restrictions with assertions.[6]

Ownership using ghost fields. The first step is quite direct. Each object has ghost field own to point to its owner. If an object o currently has no owner (as is the case when initially constructed), $o.own = \mathbf{null}$. An object encapsulates the objects it transitively owns. Transitive ownership is a relation on references, defined inductively as follows: $o \succ p$ iff either $o = p.own$ or $o \succ p.own$. Note that $\succ$ is state-dependent. The invariant, $\mathcal{I}^C$, for a class C is considered *admissible* just if whenever $\mathcal{I}^C$ depends on $p.f$ for some object p then either $\mathbf{self} = p$ or $\mathbf{self} \succ p$.

By representing the ownership relation by a ghost pointer to the owner, we have imposed the invariant that an object has at most one owner. Transitive ownership thus imposes a hierarchical structure on the heap —though one that is mutable.

Rather than preventing aliases to encapsulated reps from clients, the inv/own discipline prevents updates that falsify the invariant. For invariants that depend only on fields of $\mathbf{self}$, this was achieved by imposing on every update $E.f := E'$ the precondition $\neg E.inv$. It would be sound, but hardly practical, to impose now the precondition

$$\neg E.inv \wedge (\forall\, o \mid o \succ E \;\Rightarrow\; \neg o.inv\ )$$

so that no object with an invariant dependent on $E.f$ is packed. One reason this precondition is impractical is that the code performing the update of E would have to have ensured that many objects are unpacked, which hardly seems modular. Another reason is that if o owns p it makes no sense to unpack p unless o is already unpacked, since when it is packed o's invariant depends on p. Finally, the transitive ownership relation, $\succ$, is not attractive for direct use in reasoning and that turns out to be unnecessary.

The idea with precondition $\neg E.inv$ for an update $E.f := E'$ is that E should get unpacked before updates are performed on it. Unpacking manifests that control is crossing the encapsulation boundary for E. The discipline uses one more ghost field, $com : \mathbf{bool}$, in order to impose a discipline whereby the flow of control across encapsulation boundaries respects the current ownership hierarchy. The name stands for "committed": $o.com$ implies $o.inv$ but says in addition that o is committed to its owner and can only be unpacked after its owner gets unpacked. This idea is embodied in two additional program invariants:

$$(\forall\, o, p \mid o.inv \wedge p.own = o \;\Rightarrow\; p.com\ ) \tag{3}$$

$$(\forall\, o \mid o.com \;\Rightarrow\; o.inv\ ) \tag{4}$$

[6] Skalka and Smith [SS05] also study use-based object confinement, for different purposes.

The key consequence of these invariants is the *transitive ownership lemma*: If $o \succ p$ and $\neg p.inv$ then $\neg o.inv$. It is now possible to maintain program invariant (2) simply by stipulating for every field update $E.f := E'$ the precondition $\neg E.inv$. If the update is made in a state where for some object o we have that $\mathcal{I}[o/\mathbf{self}]$ depends on $E.f$ then $o \succ E$ by admissibility of $\mathcal{I}$. And by the transitive ownership lemma, $\neg E.inv$ implies $\neg o.inv$.

In summary, the inv/own discipline prevents interference neither by alias control nor by syntactic conditions but rather by a precondition, expressed in terms of auxiliary state that encodes dependency and hierarchy.

Typically, the precondition of a method that performs updates is $\mathbf{self}.inv \wedge \neg \mathbf{self}.com$. If it performs updates on a parameter x, an additional precondition will be $x.inv \wedge \neg x.com$. Manipulation of the com field is part of what it means to pack and unpack an object. For $\mathbf{unpack}\ E$, the stipulated precondition is now $E.inv \wedge \neg E.com$ and the effect[7] is

$$E.inv := \mathbf{false};\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = E\ \mathbf{do}\ o.com := \mathbf{false};$$

For $\mathbf{pack}\ E$, the stipulated precondition is

$$\neg E.inv \wedge \mathcal{I}^C[E/\mathbf{self}] \wedge (\forall\, o \mid o.own = E \Rightarrow o.inv \wedge \neg o.com\ )$$

where C is the type of E. The effect is

$$E.inv := \mathbf{true};\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = E\ \mathbf{do}\ o.com := \mathbf{true};$$

5. Additional aspects of the inv/own discipline

The ingredients of the discipline are

- Assertions.
- Ghost fields.[8]
- Updates to ghost fields, including update of an unbounded number of objects (in $\mathbf{pack}\ E$, for example, the com field of every object owned by E is updated).

This is quite limited machinery and thus the discipline is suitable for use in a variety of settings. It could be formalized within an ordinary program logic, most attractively a proof outline logic [PdB05b]. It is being explored in the context of Spec#, a tool based directly on a system of verification conditions, and in a tool developed by de Boer and Pierik [dBP02] (www.cs.uu.nl/groups/IS/vft/). In both cases the assertion language is (roughly) first order plus reachability but that is not essential.

Rather than relying entirely on annotations, practical use of the discipline can be streamlined through some simple abbreviations [BDF$^+$04, LM04]. A marked field declaration $\mathbf{rep}\ f : T$ is syntactic sugar for the invariant $\mathbf{self}.f.own = \mathbf{self}$ and $\mathbf{peer}\ f : T$ is syntactic sugar for $\mathbf{self}.f.own = \mathbf{self}.own$.

The discipline supports an attractive extension to frame conditions: without mention in a "modifies" clause, a method can update committed objects. For details see [BDF$^+$04].

Quantification in invariants. We have formalized the program invariants (2–4) using quantifications that range over all allocated objects. However, within declared invariants (those we name with $\mathcal{I}$) such quantification is problematic. First, if quantification ranges over currently allocated objects then a quantified formula can be falsified by garbage collection, e.g., $(\exists\, o \mid P(o)\ )$ is falsified if the only object with property P gets collected. Garbage sensitivity has been studied in depth by Calcagno et al. [COB03]. A workable solution —and the one we adopt— is to ignore garbage collection in program logic, so quantifications range over all objects that have been allocated.

This still leaves the possibility of falsification by allocation of a new object. Pierik, de Boer, and Clarke [PCdB05] have explored, for example, the Singleton pattern [GHJV95] where one might want the invariant of $Singleton$ to be

$$(\forall\, p \mid \mathbf{type}(p) \leq Singleton \Rightarrow p = Singleton.it\ ) \tag{5}$$

where it is a static field of class $Singleton$ and $\leq$ denotes subtype. (We write $\mathbf{type}(p)$ for the allocated type of an

[7] The "**foreach**" part of the effect can be expressed using a specification statement: modifies com, ensures $(\forall\, o \mid (o.own = E \wedge \neg o.com) \vee (o.own \neq E \wedge o.own = \mathbf{old}(o.own))\ )$.

[8] With inv, own ranging over values that include class names, i.e., slightly beyond ordinary program data types. Similar use of class names is available in the JML specification language via the $\mathbf{type}$ operator [LCC$^+$03].

object p.) This problem can be prevented by including as an admissibility condition that an invariant is not be falsifiable by construction of new objects. The authors of the Boogie papers [BDF+04, LM04] intend that invariants use quantification only over owned objects, which achieves this effect. Barnett and Naumann [NB04] impose it explicitly in their definition of admissibility.

An alternative is for predicates like (5) to be considered admissible and to stipulate a suitable precondition for object construction (**new**). This alternative has been worked out by Pierik et al. [PCdB05] based on a notion of update guard adapted from that discussed in Section 7.

Ownership transfer. A useful feature of the inv/own discipline is that, while it imposes hierarchical structure on the heap, that structure is mutable. Field own is initially **null**; a fresh object has no owner. The field is updated by special command $\mathbf{set\text{-}owner}\ E\ \mathbf{to}\ E'$, the effect of which is simply $E.own := E'$. As with ordinary field update, it is subject to precondition $\neg E.inv$. Moreover, in the case that $E' \neq \mathbf{null}$ the command adds to the objects owned by E' —and it adds to those transitively owned by the transitive owners of E'. Their invariants depend on their owned objects so we require them to be unpacked. The stipulated precondition for $\mathbf{set\text{-}owner}\ E\ \mathbf{to}\ E'$ is $\neg E.inv \wedge (E' = \mathbf{null} \vee \neg E'.inv)$. Thus the ownership structure can change dynamically when the relevant invariants are not in force.[9]

Change in ownership structure is difficult or impossible with ownership type systems, in part because the type system imposes the ownership conditions as a program invariant, i.e., true in every state. Transfer has been found to be useful in a number of situations. The most common seems to be initialization by the client of an abstraction that then becomes owned by another; this was pointed out by Leino and Nelson [DLN98] with the example of a lexer that owns an input stream but that stream is provided initially by the client. Transfer between peer owners is appropriate, for example, with several queues of tasks that are moved between queues for load balancing. The trickiest form of transfer is when an encapsulated rep is released to clients; this form has been highlighted by O'Hearn in the example of a memory allocator, considering that the allocator owns elements of the free list [OYR04]. Other examples can be found in [LM04, BN05b].

Transfer of ownership is important but infrequent. The Spec# project [BLS05] is exploring inference to determine where the **set-owner**, **pack** , and **unpack** commands are needed, in order to lighten the annotation burden. Integration with ownership types merits investigation for this purpose.

Taking subclasses into account. If C is a subclass of D then an instance of C has fields of D and of C. Moreover, it should maintain the invariant, $\mathcal{I}^D$, of D but C may impose an additional invariant $\mathcal{I}^C$. Instead of using a boolean to track whether "the" invariant is in effect, the general form of the inv/own discipline lets inv range over classnames, with the interpretation that $o.inv \leq C$ means that o is packed with respect to the invariant of C and of any superclasses of C. This works smoothly if we assume $\mathcal{I}^{Object}$ is **true**.

Owned objects are now owned at a particular class, i.e., field own ranges over **null** and pairs (C, o) with $\mathbf{type}(o) \leq C$ indicating that the object is owned by o at class C and is part of the representation on which $\mathcal{I}^C$ (and invariants in subclasses) depends.

The **pack** and **unpack** commands are revised to mention the class involved. For $\mathbf{unpack}\ E\ \mathbf{from}\ C$, the stipulated precondition is now $E.inv = C \wedge \neg E.com$ and the effect is

$$E.inv := super(C);\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = (E, C)\ \mathbf{do}\ o.com := \mathbf{false};$$

For $\mathbf{pack}\ E\ \mathbf{to}\ C$, the stipulated precondition is

$$E.inv = super(C) \wedge \mathcal{I}^C[E/\mathbf{self}] \wedge (\forall\, o \mid o.own = E \Rightarrow o.inv \wedge \neg o.com\ )$$

and the effect is

$$E.inv := C;\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = (E, C)\ \mathbf{do}\ o.com := \mathbf{true};$$

The program invariants are also adapted slightly, as follows.

$$(\forall\, o, C \mid o.inv \leq C \Rightarrow \mathcal{I}^C[o/\mathbf{self}]\ )$$
$$(\forall\, o, p, C \mid o.inv \leq C \wedge p.own = (o, C) \Rightarrow p.com\ )$$
$$(\forall\, o \mid o.com \Rightarrow o.inv \leq \mathbf{type}(o)\ )$$

Methods are dynamically dispatched, which raises the question how to express the precondition that before was

[9] Because field own is mutable, it is possible to create a cycle of owners. But owing to the stipulated preconditions of the discipline, objects in a cycle cannot be packed.

$$
\begin{array}{l}
\textbf{class } Subject2 \ \{ \ //\text{Alternate version} \\
\quad \textbf{private rep } x : Integer := \textbf{new } Integer(0); \\
\quad \textbf{private rep } z : \textbf{int} := 0; \\
\quad \textbf{invariant } \mathcal{I}^{Subject2'} \quad \text{where } \mathcal{I}^{Subject2'} =_{df} 0 \leq z \\
\quad \textbf{method } m() \ \{ \ x.incr(); \ \} \quad \}
\end{array}
$$

Fig. 5. Revised $Subject2$.

just $inv = \mathbf{true}$. Now an implementation in class C needs precondition $inv = C$. The Boogie paper [BDF$^+$04] introduces notation which at a method call site means $E.inv = \mathbf{type}(E)$ but in the method implementation means that $\mathbf{self}.inv$ equals the static type. This is worked out by treating method inheritance as an abbreviation for a stub method with appropriate **unpack** and **pack**; this generates a proof obligation on the inherited method: it must preserve any invariant that is introduced in the subclass.

Soundness and completeness. Soundness of the discipline is taken to mean that the three displayed conditions hold in every reachable state of a *properly annotated program*, i.e., one in which every field update and every instance of a special command **pack**, **unpack**, or **set-owner** is preceded by an assertion that implies the stipulated precondition.

For sequential programs in a Java-like language, soundness is sketched in the original Boogie paper [BDF$^+$04] and more rigorously in [NB06]; see also [LM04]. Extension of the discipline to concurrent programs has also been investigated [JLS04].

Completeness is another matter. It is not clear to this author how to formulate an interesting notion of completeness. Clearly it has to be relative to completeness of an underlying proof system. The discipline hinges on having every object invariant expressed in the form $inv \Rightarrow \mathcal{I}$ with $\mathcal{I}$ admissible. Does completeness say that every predicate of this form that is in fact invariant can be shown so in a proof outline following the discipline? There are related questions: Which admissible predicates are expressible as formulas? Which formulas denote admissible predicates? A convincing notion of completeness would be especially useful if it could be adapted to other disciplines like the one discussed in Section 7.

In what sense are invariants necessary at all? One could perhaps simply conjoin (2), (3), and (4) to preconditions and postconditions throughout the program. But this raises another expressiveness question. And for modularity it might require abstraction from internals, e.g., using model fields. Notions of completeness that take modularity into account have recently been studied by Pierik and de Boer [PdB05a].

Static invariants. We have focused on object invariants that depend on instance fields. It is also sensible for an object invariant to depend on static fields, e.g., the Singleton invariant (5). There is also the possibility of a static invariant for a class. Examples are given by Leino and Müller [LM05] and by Pierik et al. [PCdB05]. The basic idea is to use a static field in the same way as inv, to represent whether the invariant of a class is in force. There are intricacies due to the way in which classes are initialized in Java.

6. Coupling invariants and simulation

This section describes, without much detail, how the inv/own discipline has been adapted to reasoning about simulations.

Fig. 5 shows an alternate implementation of class $Subject2$ from Fig. 4. In this section, we assume that $Subject2$ in Fig. 4 has been corrected by deleting method $leak$. The behavior of the two versions is the same, at the level of abstraction of the programming language, e.g., ignoring speed and size of object code. (More formally, the two versions give rise to equivalent behavior in any well formed context of usage.) This would still be true if, e.g., a method is added to read the current integer value of x. The standard way to prove behavioral equivalence of two modular units such as classes is by means of a coupling relation that has the simulation property. A *coupling* relates states for one implementation with states for the other. For an instance s of $Subject2$ in the first version (Fig. 4) and s' for the second version (Fig. 5), a suitable coupling is

$$s.x.val = s'.x.val \wedge s.y.val - s.x.val = s'.z$$

Such a relation is a *simulation* provided that it is preserved by corresponding method implementations —as it is by the two versions of m in the example. (The same technique is also used to prove refinement: in case one implementation diverges less often or is less nondeterministic, the notion of preservation is adapted slightly. Another use is to

justify "observationally pure" method calls in assertions [BNSS06, Nau06b].) The technique is practical because the simulation property only needs to be proved for the re-implemented methods: For arbitrary program contexts, simulation should follow from simulation for the revised class, by a general *representation independence* property of the language.

The simulation technique was articulated by Hoare [Hoa72] drawing on work of Milner [Mil71] and has seen much development for use with purely functional programs [Plo73, Mit86, Pit00] as well as first order imperative and concurrent programs [LV95]. For first order imperative programs the topic is thoroughly surveyed in the textbook by de Roever et al. [dRE98]. Object oriented programs have features in common with higher order imperative programs, for which representation independence is nontrivial owing to semantic difficulties [OT95, Pit97, Nau02]. Two sources of complication in object oriented programs are inheritance and the ubiquitous use of recursive classes; these were addressed by Cavalcanti and the author [CN02] —under the drastic simplification that copying is used instead of sharing. Their results have been used to validate laws of program refactoring [BSC03, BSCC04].

The representation independence property, i.e., the possibility of reasoning in a modular way using simulations, is a measure of the encapsulation facilities of a language. We have seen how reentrant callbacks and heap sharing pose a challenge for encapsulation in object oriented programs. Using a static analysis for alias control in order to impose an ownership structure just for the class under revision, Banerjee and Naumann [BN05a] prove representation independence for a rich imperative fragment of Java with class-based visibility, inheritance and dynamic binding, type casts and tests, recursive types, etc. A key feature of that work is the notion of *local coupling* which is a binary relation not on complete program states but just on a fragment of the heap consisting of a single instance of the class under revision together with its reps. That is, a local coupling relates pairs of islands. This induces a coupling relation for the entire program state.

There are two main shortcomings to the early work of Banerjee and Naumann [BN05a]. First, ownership transfer is disallowed by their confinement rules. Second, the result is inadequate for programs with callbacks because it is in terms of the standard notion of simulation: for method m to preserve the coupling means that if two states are initially coupled, then running the two versions of the implementation of m leads to coupled states. Recall that representation independence says, with A the class for which two versions are considered, that if all methods m of class A have the simulation property then the relation is preserved when those methods are used in arbitrary program contexts. In fact the proof obligation is not simply that m preserves the coupling, but rather that it preserves the coupling *under the hypothesis that any method m invokes preserves the coupling.*[10] This assumption can be useful in establishing the simulation property for m, but only if the two implementations make the same method call and from a state where the coupling holds. But at intermediate steps in paired invocations of (the two versions of) m, the coupling relation need not hold —essentially for the same reason as invariants need not hold during updates of local state. The hypothesis is of no help if a client method is invoked at an intermediate step where the coupling does not hold; so the result is sound and even useful but the applications in the cited work rely on ad hoc reasoning to establish the simulation property.

It turns out that the inv/own discipline, which is concerned with preservation of invariants, can be adapted to simulations, i.e., preservation of coupling relations; see [BN05b]. The intuition is that a coupling is just an invariant over two copies of program state. Moreover, field inv is observable (by specifications), so both versions of a method m of A have the same $\mathbf{unpack}/\mathbf{pack}$ structure. Thus the coupling can take the form of an implication with antecedent inv. This form of coupling can then hold at intermediate points in m, in particular at method calls —so the hypothesis is now of use.

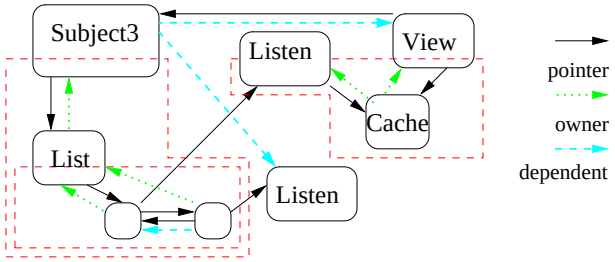
The adaptation is not trivial because the inv/own discipline only controls updates. Stronger encapsulation is needed for representation independence than for invariants. Recall the example $leak$ in Section 4. If we revise it as follows, so that the leaked reference is only read, then the program is compatible with the inv/own discipline.

$$\begin{array}{l}\mathbf{class}\ Main\ \{\quad s : Subject2 := \mathbf{new}\ Subject2;\\ \quad \mathbf{method}\ main()\ \{\ i : Integer := s.leak();\ Print(i.val);\ \}\quad \}\end{array}$$

For invariants, it is only a problem if i is updated. But for simulations, we need independence from reps —not even dependence by reading— as otherwise a client's behavior can be affected and the representation is not fully encapsulated. This can be achieved by stipulating additional preconditions for field access [BN05b] (which in practice can usually be discharged trivially in virtue of standard visibility rules).

Informal considerations of information hiding suggests that clients should not read fields of reps, and this is confirmed by the analysis of representation independence. In this light, it seems that the main advantage of the inv/own

[10] 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.



$$\begin{array}{l}
\textbf{class } \mathit{Subject3}\ \{\quad \mathit{val} : \textbf{int};\quad \mathit{vsn} : \textbf{int};\quad \mathit{listeners} : \mathit{List}\langle \mathit{Listener}\rangle;\\
\ldots\}\\
\textbf{class } \mathit{Cache}\ \{\quad \mathit{vsn} : \textbf{int};\quad \mathit{val} : \textbf{int};\quad \}\\
\textbf{class } \mathit{View}\ \{\quad \mathit{sbj} : \mathit{Subject};\quad \textbf{rep } \mathit{st} : \mathit{Cache};\\
\quad \textbf{invariant } \mathcal{I}^{\mathit{View}}\\
\qquad \text{where } \mathcal{I}^{\mathit{View}} =_{\mathit{df}} \mathit{sbj}.\mathit{vsn} - 1 \le \mathit{st}.\mathit{vsn} \le \mathit{sbj}.\mathit{vsn} \wedge (\mathit{st}.\mathit{vsn} = \mathit{sbj}.\mathit{vsn} \Rightarrow \mathit{st}.\mathit{val} = \mathit{sbj}.\mathit{val});\\
\ldots\}\\
\textbf{class } \mathit{Listener}\ \{\quad \mathit{st} : \mathit{Cache};\\
\quad \textbf{method } \mathit{notify}()\ \{\ \ldots\ \}\quad \}
\end{array}$$

Fig. 6. Observer pattern using separate listeners.

discipline over ownership types is the ability to temporarily violate the ownership property in order to transfer objects between owners.

7. Beyond single-object invariants

At the beginning of Section 2 we focused attention on situations where each instance of a class is intended to provide some cohesive abstraction such as a collection. Such examples are ubiquitous, but so too are situations where several objects cooperate to provide some abstraction. This section sketches an extension of the inv/own discipline to one form of cooperation.

One example is iterators. To equip a Collection with the possibility of enumerating its elements, a separate object is instantiated for each enumeration. These Iterator objects need access to the internal data structure of the Collection, to get elements of the Collection and to track whether the Collection has changed in a way that makes the Iterator inconsistent and unusable.

One can imagine formulating a single invariant that pertains to the collective state of a Collection and its Iterators, but it is not clear with what program structure this invariant would be associated. Perhaps the iterator and Collection classes could be put in a single module, but associating the invariant with the module does not reflect that the natural unit is a single Collection instance together with its iterators.

An alternative using more familiar notions is to express the conditions in the object invariant for an Iterator. But it is not feasible for an Iterator to own the Collection on which it depends, since Iterators serve as part of the interface to clients. Aldrich and Chambers [AC04] explore a flexible notion of ownership type where the dominator property is not necessarily imposed, but absent this property it is not clear what modular reasoning is supported.

The need for object invariants to depend on non-owned objects arises in quite simple situations. In Section 2 we considered the $Solver$ invariant that involves doubly-linked list conditions $p.next = \mathbf{null} \vee p.next.prev = p$ for all p in $NS.next^*$. It is possible to associate the entire invariant with class $Solver$, but at the cost of a quantifier and reasoning about reachability. A less centralized formulation would push some of the conditions into object invariants for the rep objects, e.g., each node could maintain the invariant $next = \mathbf{null} \vee next.prev = \mathbf{self}$. But for this to be admissible, a node would need to own its successor. Such an ownership structure is workable for acyclic doubly-linked lists but not for cyclic ones (and it is awkward if iteration is used instead of recursion).

As a more elaborate example, consider the variation on the Observer pattern depicted in Fig. 6, where a separate Listener object is the target of the $notify$ callback. The dashed and dotted arrows are explained in due course. Dashed rectangles are used as before to indicate ownership encapsulation. In this arrangement it would seem that both the Listener and the View need to read and update their shared Cache object. The situation is similar to that for Collections/Iterators. We return to this point later. The next point to consider is that we aim to specify that notifications are

required: the Subject has a version number that is incremented each time it is updated, and $notify$ brings the View back in sync. For simplicity we treat the state of the Subject as an integer, val. The View maintains a copy of the state of the sensor, with its version number, in its Cache object. View also maintains the invariant that this version is not more than one step behind. We assume it is untenable for the View to own its Subject sbj. So this invariant is inadmissible according to the previous definition, because it depends on fields val and vsn of sbj.

A prerequisite for this dependence is of course that those fields are visible in $View$. Rather than giving them public visibility, let us suppose that $Subject3$ includes an explicit declaration

$$\textbf{friend}\ View\ \textbf{reads}\ vsn, val;$$

to extend the scope of visibility. The intention is not only to broaden the scope (as little as possible) but also to license dependence of $\mathcal{I}^{View}$ on these fields. It is thereby also signalled to $Subject3$ that it has a proof obligation: if s has type $Subject3$ then updates to $s.val$ and $s.vsn$ must not falsify the invariant of any object v of type $View$ that is dependent on s.

Visibility based invariants. Müller and others [Mül02, LM04, MPHL04] have worked out sound rules for reasoning about invariants in this sort of *peer* relationship. They use the term *visibility based invariant*, in contrast to ownership based invariants. For our example, the idea would be that $Subject3$ is responsible to maintain any invariants visible to it. In some examples this is quite manageable, but in general there is a problem. The visibility based approach works at the level of classes. In reasoning about an update of a given instance s of $Subject$, one must consider the invariant of any instance v of $View$ since the invariant of $View$ depends on fields of $Subject$. The question is how the reasoner gets a handle on those objects, given that there can be many instances of $View$, dependent on many different instances of $Subject3$.

In the example at hand, $s.listeners$ is intended to hold references to all listeners for views dependent on s, so that they can be notified of updates. Suppose that listeners have a field $myview$ so that the views dependent on s are those in $s.listeners.next^*.myview$. Then it suffices to prove that updating $s.val$ or $s.vsn$ does not falsify the invariant of those views. Thus the precondition for $s.val := \ldots$ would say that the dependent views are unpacked:

$$\mathbf{self}.inv > Subject \wedge (\forall\, v \mid v\ \mathbf{in}\ s.listeners.next^*.myview \Rightarrow v.inv > View\ ) \tag{6}$$

It is certainly possible to establish this precondition. In order to update fields declared in $Subject$, s must be unpacked from $Subject$, so s is not committed. If the views have the same owner, they also are not committed and thus they can be unpacked if they are not already. But must they have the same owner?

Packing and unpacking are designed to embody hierarchical encapsulation. The example involves peers that are in some sense within the same encapsulation boundary. Moreover, to repack a view v, the $Subject$ would need to justify that $\mathcal{I}^{View}[v/\mathbf{self}]$ holds —but why should $\mathcal{I}^{View}[v/\mathbf{self}]$ be visible in $Subject$?

The preceding challenges are not insurmountable using just standard proof rules and the visibility assumptions. But by using a ghost field to track the relevant dependencies, more localized reasoning can be achieved.

The friendship discipline. We posited temporarily that an instance s of $Subject3$ has access to its dependent views via $listener$ and $myview$, but in fact $Listener$ has no such field —so Subject3 only has references to the $Listener$s on which it is supposed to invoke $notify$. For another example of such a situation, a long-lived Collection might have many associated Iterators. The Iterators could depend on a timestamp field in the collection, in order that an Iterator can be considered invalid if the collection gets updated. But there may be no reason for the Collection to maintain a list of its iterators. Instead of incurring a performance cost to maintain the list merely for the sake of reasoning, it can be stored in a ghost field.

The friendship discipline [BN04] extends the inv/own discipline by adding a ghost field $deps$ to hold references to dependents (the dashed arrows in Fig. 6). As before, consider a declaration "$\textbf{friend}\ View\ \textbf{reads}\ vsn, val;$" in $Subject3$. We use the terminology *granter* for class $Subject3$ and *friend* for the class $View$ to which access is granted. Here access means that the admissibility condition is relaxed to allow the invariant of class $View$ to depend on vsn and val in $Subject3$. Moreover each instance v of $View$ is required to maintain the following invariant:

If in the current state $\mathcal{I}^{View}[v/\mathbf{self}]$ depends on s then $v \in s.deps$.

One can now adapt the precondition (6) for field update to quantify over just $s.deps$. Special commands $\mathbf{attach}$ and $\mathbf{detach}$ are used to manipulate $deps$, much like $\mathbf{pack}$ and $\mathbf{unpack}$ [BN04, NB06].

The friendship discipline also rectifies another flaw of (6). Instead of requiring that all dependent views are unpacked, we account for the possibility that the update is not going to falsify the invariant of a packed view. For

example, suppose that we dropped the requirement, $sbj.vsn - 1 \leq st.vsn$, that a $View$ not lag too far behind, keeping as invariant only this:

$$st.vsn \leq sbj.vsn \wedge (st.vsn = sbj.vsn \;\Rightarrow\; st.val = sbj.val) \tag{7}$$

Then an update of the form $vsn, val := vsn + 1, \ldots$ never falsifies the invariant of a view.

More generally, we allow $View$ to declare conditions —visible to the granting class $Subject3$— under which its invariant is not falsified. The declaration

$$\textbf{guard } sbj.vsn := \alpha \textbf{ by } U \quad \text{where } U =_{df} \alpha - 1 \leq \textbf{self}.st.vsn \leq \alpha$$

protects the original invariant $\mathcal{I}^{View}$ including condition $sbj.vsn - 1 \leq st.vsn$. (Here α is a fresh variable to be instantiated as needed.) This is because the proof obligation imposed on $View$ for U is satisfied:

$$\mathcal{I}^{View} \wedge U \;\Rightarrow\; wp(\textbf{self}.sbj.vsn := \alpha)(\mathcal{I}^{View})$$

Owing to this we can now weaken the precondition (6) for field update, since under condition U the invariant of a packed view cannot be falsified. For update $vsn := vsn + 1$ in code of $Subject3$, the precondition is [11]

$$inv > Subject3 \wedge (\,\forall\, v \mid v \textbf{ in } deps \;\Rightarrow\; v.inv > View \;\vee\; U[\textbf{self}/sbj, v/\textbf{self}, (vsn+1)/\alpha]\,) \tag{8}$$

Just as, for any object o, the field $o.inv$ serves as a publicly visible abstraction of $\mathcal{I}[o/\textbf{self}]$, here U serves to abstract from $wp(\textbf{self}.sbj.vsn := \alpha)(\mathcal{I}^{View})$ in a way suitable to be visible in $Subject$, without fully revealing $\mathcal{I}^{View}$. The substitutions adapt the update guard from the nomenclature of $View$ to that of $Subject3$ and to the particular update $vsn := vsn + 1$.

History constraints. For the invariant (7), an alternative to the friendship discipline is to use history constraints [LW94]. A history constraint is a two-state predicate on an object, interpreted as a constraint on any two successive visible states of the object (e.g., states at method call or return). Let us use primes on field names to designate the "after" state, to give an example history constraint that is satisfied by $Subject3$:

$$vsn \leq vsn'$$

That is, vsn increases monotonically. Invariant (7) cannot be falsified by any update of vsn that satisfies the constraint.

In general, if the granter declares a history constraint and the friend's invariant is not falsifiable by updates satisfying the constraint then no precondition concerning the friend needs to be imposed on updates by the granter. To the author's knowledge, history constraints have only been studied in the case where they depend on the object's own fields, not on fields of reps [LW94]. It could be valuable to study constraints that depend on fields of reps (just as our example update guard depends on fields of the Cache of $View$).

A shortcoming of history constraints is that their meaning depends on a notion of visible state, just like the visible state semantics of invariants. The inv/own discipline dodges this by using field inv to maintain program invariants which are true "at every semicolon". Perhaps there is a comparable notion of history constraint.

It is not clear how to use a history constraint if $\mathcal{I}^{View}$ includes the condition $sbj.vsn - 1 \leq st.vsn$ which we use to force notifications. It is true that the vsn field of a Subject is incremented by one in each atomic update, but the strongest history constraint is that it is nondecreasing, since at some computation steps it is unchanged and after sufficiently many steps it can change by more than one.

An advantage of history constraints is that they handle a sequence of multiple updates to $Subject$ whereas the update guard is formulated in terms of an atomic update. In fact this can also be achieved by slightly extending the friendship discipline; this has been worked out and applied to the iterator pattern [NB06]. The key change (from [BN04]) is to allow a friend's invariant to depend on the inv field of the granter. An invariant of the form $sbj.inv \;\Rightarrow\; \ldots$ is not falsified by unpacking sbj, which is needed anyway in order to perform updates. Then the friend's update guard for inv imposes a proof obligation only when the granter is re-packed.

Update/yielding. How can a granter establish the U case in precondition (8)? To reason that a given view satisfies U, in the context of $Subject$, it might be possible to use specifications of methods of $View$. In particular, U could be given as postcondition of $notify$.

A history constraint is something like a pre/post specification that applies not to a particular method or command but to arbitrary pairs of observations. One can see an update guard as a precondition for arbitrary steps; what about

[11] For clarity we use substitution notation but a precise formulation must handle aliasing in some way, e.g. [AO97].

a postcondition thereof (in addition to the invariant)? Under precondition U, increment of $sbj.vsn$ by one yields a state where the view's version lags exactly one step behind. This can be declared as a postcondition in the **guard** declaration; the idea is worked out in [BN04].

Friendship for coupling. There is a practical problem with the use of simulation to prove equivalence and refinements: Few tools exist for proving the simulation property for two versions of a method implementation in a language like Java. There is, however, a technique sometimes called Reynolds' method for converting a simulation property into an ordinary correctness property [Rey81, dRE98, Gri93]. A disjoint copy of the state space is introduced and the two method bodies to be related, say S_0 and S_1, are composed into the single command $S_0; S_1'$ where S_1' is the same as S_1 but acting on the copy. Making a disjoint copy is easy in the case of ordinary variables, where suitably named fresh variables can be used. For example, if both programs act on a client variable x then S_1' is changed to act on x' and a suitable coupling includes the condition $x = x'$. The author has extended this method to programs acting on mutable heap objects (a special case is presented in [Nau06a]). Whereas primitives can be related as in $x = x'$, heap objects are duplicated and corresponding pairs must be tracked [BN05a, BN05b]. Two heaps are encoded in a single heap using a boolean ghost field $dashed$ to distinguish objects in one heap from those in the other. A pointer-valued ghost field, $mate$, is used so that a coupling can require corresponding objects to be related by $x.mate = x'$ and $x = x'.mate$.

Since a coupling is but an invariant over a doubled state space, one would hope to express encoded couplings using extant techniques for invariants. The condition $x.mate = x'$ and $x = x'.mate$ is not admissible as an ownership-based invariant but it is admissible using friendship.

8. Making footprints explicit

This section contrasts the inv/own discipline with two alternative approaches that rely on more powerful logics for assertions. The theme in these approaches is to make the "footprint" of an invariant more explicit; they require and facilitate reasoning about the footprint of client programs and its disjointness from the footprints of invariants.

The inv/own discipline requires clients to establish an object's invariant before invoking methods on it, but this is made easy because the invariant effectively has the form $inv \Rightarrow \mathcal{I}$. Although $\mathcal{I}$ is typically not visible to clients, they can establish $\neg inv$ by unpacking the object or maintain inv by relying on postconditions of the object's methods. Hiding of $\mathcal{I}$ is not an intrinsic part of the approach; rather the approach is designed to facilitate hiding.

Kassios [Kas06b, Kas06a] makes more explicit use of visibility. An object invariant is treated as a model field [LCC+03] of boolean type. The field name is public and can be used in method specifications, but its definition (in terms of encapsulated state) is only visible within the object's class. This achieves some of the effect of inv. Similarly, Parkinson [BP05, Par05] gives a logic in which an invariant can be treated as a named predicate, with the name visible for use in public method specifications but the definition only visible within the class. (Such method specifications can be seen as using an existentially quantified predicate, witnessed by a definition inside the class [BTS05].) In both approaches, clients are responsible for establishing an object's invariant before invoking methods on it.

Clients are also responsible for not falsifying the invariant of an object, in between invocation of its methods. The inv/own discipline uses ownership to encode the "footprint" of an invariant, i.e., the locations on which it currently depends. (The term "location" is used here deliberately; the footprint is a set of individual field locations, i.e., object references paired with field names.) The rule for field update ensures that client does not step on the footprint when the invariant is in force.

In addition to using a model field to represent an invariant, Kassios uses a model field to represent the footprint, called a dynamic frame, of the invariant. (The technique can be used with other model fields as well.) The condition that field F contains all locations on which $\mathcal{I}$ depends is expressed by a second order predicate "F frames $\mathcal{I}$" (defined by quantifying over all global states). Together with other special expressions, this provides for elegant specifications that allow clients to reason about disjointness of their updates from the footprint of an invariant, even though neither the invariant nor the contents of the dynamic frame are exposed outside their appropriate scope.

The technique of Kassios is quite flexible and has been applied to multi-object invariants including a general form of the Iterator pattern [Kas06b].

Parkinson [Par05] treats a fragment of Java using the opaque ("abstract") predicates of Bierman and Parkinson [BP05] for a version of separation logic. The special predicate $o.f \mapsto v$ expresses that o is an allocated object and the value of its field f is v. The separated conjunction $P * Q$ of predicates expresses that the heap can be partitioned into some locations that support the truth of P (e.g., $o.f$ supports the truth of $o.f \mapsto v$), some disjoint ones that sup-

port Q and possibly others.[12] Footprints are first-class entities in the work of Kassios; in separation logic the notion of footprint is implicit but transparent. The typical form for a method precondition is $Inv * P$ where Inv is the (name of) the object invariant and P is a predicate on some objects provided by the client.

A key feature of separation logic is the *frame rule* which infers $\{P * R\}S\{Q * R\}$ from $\{P\}S\{Q\}$. In particular, if S is a client program that can be proved to satisfy $\{P\}S\{Q\}$ and R is an invariant with footprint disjoint from P then S does not falsify it. For single-instance modules, the fact that an invariant is disjoint from objects accessed by a client, and not visible to the client, can be expressed by a higher order frame rule [OYR04]. This does not work for dynamically allocated objects but Parkinson achieves a similar effect using opaque predicates.

The frame rule is sound in virtue of the *frame property* of programs acting on the heap (emphasized by O'Hearn and Yang [YO02]). First, any terminating computation of a command S reads and writes some finite subset of the initially-existing locations (as well as new ones it allocates). Moreover, if the initial heap decomposes as a disjoint union $h \cup h'$ where h contains all the locations on which S acts, then the final heap decomposes as a disjoint union $h'' \cup h'$ —that is, h' is untouched. Soundness of the frame rule relies on the "tight interpretation" of correctness statements used in separation logic: $\{P\}S\{Q\}$ means that the footprint of P covers all (initially allocated) locations on which S acts. Thus, by the frame property of S, $\{P\}S\{Q\}$ implies $\{P * R\}S\{Q * R\}$ for any R, because the precondition means that the footprint of predicate R is disjoint from the footprint of command S.

The frame property can be made more explicit using notation that treats the correctness statement as a type for S:

$$S : \{P\} - \{Q\} \tag{9}$$

(For S to have the type $\{P\} - \{Q\}$ just means the triple $\{P\}S\{Q\}$ is valid.) Birkedal et al. [BTSY05] give a type system in this style, treating $\{P * R\} - \{Q * R\}$ as a subtype of $\{P\} - \{Q\}$. Their system comes close to making explicit that (9) amounts to the following, using a quantified type.

$$S : (\forall\, R \mid \{P * R\} - \{Q * R\}\ ) \tag{10}$$

If a client program S can be proved to satisfy some correctness statement $\{P\}S\{Q\}$, which perhaps mentions some opaque predicates that name invariants of objects of interest to S, the displayed property tells us that S cannot falsify invariants of objects not in its footprint.

With this in mind, the inv/own discipline can be described as follows. If S is properly annotated then it has the property

$$S : (\forall\, I \mid \{I\} - \{I\}\ ) \tag{11}$$

where I ranges only over predicates with footprint explicitly encoded by ownership. (The papers on inv/own only show the property with I ranging over explicitly declared invariants.) Note that this is much weaker than (10) which says S preserves all predicates disjoint from the footprint of S. On the other hand, (10) comes at the cost of requiring a sufficiently strong specification $\{P\} - \{Q\}$ to capture the footprint of S. Without that, nothing can be said about what invariants are preserved by S. By contrast, (11) requires only proper annotation, which can be far less than full functional specifications.

Another difference between the inv/own discipline and separation logic is that in the latter, the frame property (10) is a consequence of a pre-post property $\{P\}S\{Q\}$ whereas (11) depends on proper annotation of S at intermediate points. The former is purely a property of the pre-post semantics of S whereas the latter involves more.

Although there is no technical notion of ownership in separation logic, some specifications lend themselves to an informal reading in terms of ownership; the slogan is that "ownership is in the eye of the asserter" [O'H05].

A complication in Parkinson's approach is that to deal with subclassing, the abstract predicates actually need to be predicate families, indexed by type. Another shortcoming is due to the fact that the separating conjunction, $*$, expresses complete disjointness. Parkinson notes that this prevents handling common idioms like the iterator pattern where controlled sharing is needed. To address this shortcoming, Bornat et al. [BCOP05] devise variations that allow overlapping heaps with read and write "permissions"; the idea is extended to Java by Parkinson [Par05]. But this idea is at an early stage of development and seems rather complicated for what is achieved.

A relatively minor difference is that Kassios and Parkinson both deal with framing at the level of the individual location, i.e., field of an object. The inv/own discipline treats footprints at the granularity of object references only. An additional mechanism like data groups [LPHZ02, LCC+03] is needed for separating the update of fields within an object.

[12] In some versions of separation logic there are no others [Rey02], but because Java is garbage collected the "intuitionistic" version is needed, in which $o.f \mapsto v$ means that the heap contains at least $o.f$ but possibly other locations.

Concerning simulation, the dissertation of Kassios encompasses refinement of class implementations but focuses on framing as a technique. The formalization does not lend itself to general results like representation independence or refactoring laws. But in light of the results discussed in Section 6 it seems likely that a useful simulation theory could be based on the approach. There has also been work on simulation in separation logic, for simple imperative programs. There are difficulties with forward simulation, because the standard semantics relies on allocation being unboundedly nondeterministic in order to validate the frame rule [MTSO04, MY05].

The approaches of Kassios and Parkinson both pay a price in going beyond a first-order assertion language. Far fewer tools can be used off the shelf and in particular less automation may be feasible. In the case of separation logic, for lack of a complete proof system new notations and results may be needed for new applications (though this shortcoming can be addressed by moving to higher order separation logic [BTSY05, BBTS05]). Although both Kassios and Parkinson apply their theories to a few interesting examples, neither approach has been explored thoroughly. Both have very interesting features and deserve to be implemented in prototype tools to facilitate more extensive experimentation.

9. Challenges for future work

We have not exhausted the issues brought up by the last example in Section 7. The guard U depends on owned objects (the $Cache$) of View, exposing some of the internal state of a view to its $Subject3$. Moreover the $Listener$ could well update the cache; indeed one could imagine that $Listener$ maintains an invariant similar to $\mathcal{I}^{View}$. In Fig. 6 we draw common ownership arrows from the cache to hint that, as in the case of a Collection/Iterators, the situation seems to be one where multiple objects comprise the public interface for an abstraction and have shared access to the reps.

Such elaborate patterns have motivated proposals for increasingly complicated ownership type systems and may well necessitate more complicated versions of the inv/own and friendship disciplines. We mention one way in which the friendship discipline, as currently formulated [BN04, NB06], is inadequate. Consider a variation on $Subject3$ where its state is not just the integer, val, but rather some data structure; then $\mathcal{I}^{View}$ would depend not on $sbj.val$, but on locations reached by paths $sbj.f.g\ldots$ into that data structure. With ownership, the admissibility condition requires that each of sbj, $sbj.f$, $sbj.f.g$ etc. is owned. Friendship instead imposes mutual obligations; intricate conditions are apparently needed to extend friendship to handle longer paths, owing to the various possibilities of sharing.

While incremental extensions can be made to address this inadequacy of the friendship discipline, what really seems to be needed is a general setup for such disciplines. While ownership is widely applicable and provides a strong form of encapsulation at fairly low cost, attempts to extend it to multiple owners or cooperating peers seem more specialized. For example, friendship caters to the situation where one instance of the granter class is depended on by multiple instances of a single other class, the friend. What about situations where several objects of the same class, or of several different classes, are interdependent and collectively provide some abstraction?

Packages are not the answer because a package is a collection of classes and does nothing to describe the configurations in which instances are intended to be deployed. What we seek is a notation in which a design and reasoning pattern can be expressed. A design pattern typically involves a configuration of instances (e.g., subject and view, collection and iterators) with certain operations and protocol. A pattern-specific discipline for reasoning could be based on a single invariant for the pattern's object configuration, expressed with the help of ghost fields to encode the configuration[13] and its protocol; or perhaps the invariant can be decentralized into interdependent object invariants.

Pattern-specific rules would need to be given —stipulated annotations for critical operations including updates of ghost variables to track the structure of interest. Verification of the pattern would involve establishing designated program invariants as a consequence of the stipulated annotations. The hope is that design patterns can be endowed with reasoning disciplines embodied by first order conditions amenable to automated theorem proving, with axioms like (2–4) that facilitate local reasoning. Justification of such a discipline likely requires interactive proof and higher order logic since it involves all programs that fit the pattern.

About the friendship discipline, Tony Hoare asked "Would it not be better to define a general facility for the user to introduce ghost variables and assertions, rather like aspects in aspect-oriented programming?"[14] Separation logic offers notation that can transparently depict groups of objects and their interrelation. But in separation logic, quantification over predicates is needed for interesting specifications, in part because patterns of heap structure are expressed using separation at the level of predicates. Why not *expressions* describing regions? Pattern matching for such expressions

[13] The Aldrich-Chambers system might help here [AC04].
[14] Personal communication, April 2004.

has been given a semantic foundation [Nau01a, Nau01b] but not thoroughly investigated. Kassios [Kas06a] takes a step in this direction by using ghost fields in a general way to hold sets of objects/locations.

A less speculative question to be investigated concerns the requirement, in separation logic, that invariants be *precise* predicates, i.e., supported by a definite region of the heap [OYR04]. In simple cases, invariants are precise in virtue of being formulated by reachability in some data structure. Ghost structure may offer a scalable and precise shadow of encapsulation.

Acknowledgements. This paper is based on one that appeared in the proceedings of FMCO 2005 [Nau05a]. A number of changes have been made; the major difference is the addition of Section 8. This version reflects feedback from the FMCO meeting and anonymous reviewers for the proceedings, from participants in the Verifying Software grand challenge especially at the Zürich meeting in October 2005, and from anonymous reviewers for Formal Aspects of Computing. Discussions with Anindya Banerjee, Mike Barnett, Frank de Boer, Ioannis Kassios, Peter Müller, Peter O'Hearn, and Cees Pierik were particularly helpful.

References

[AC04] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 1–25, 2004.

[AO97] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2 edition, 1997.

[BBTS05] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *European Symposium on Programming (ESOP)*, volume 3444 of *LNCS*, pages 233–247, 2005.

[BCOP05] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 259–270, 2005.

[BDF+04] 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(6):27–56, 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, 2002.

[BLS03] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 213–223, 2003.

[BLS05] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, International Workshop (CASSIS 2004), Revised Selected Papers*, volume 3362 of *LNCS*, pages 49–69, 2005.

[BN02] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 166–177, 2002.

[BN04] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 54–84, 2004.

[BN05a] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, November 2005. Extended version of [BN02].

[BN05b] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 387–411, 2005.

[BNSS04] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004. Technical Report NIII-R0426, University of Nijmegen.

[BNSS06] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. Allowing state changes in specifications. In Günter Müller, editor, *International Conference on Emerging Trends in Information and Communication Security (ETRICS)*, volume 3995 of *LNCS*, pages 321–336. Springer, 2006. Extended version of [BNSS04].

[BP05] G.M. Bierman and M.J. Parkinson. Separation logic and abstraction. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 247–258, 2005.

[BSC03] P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A refinement algebra for object-oriented programming. In L. Cardelli, editor, *European Conference on Object-oriented Programming (ECOOP)*, number 2743 in LNCS, pages 457–482, 2003.

[BSCC04] P. H. M. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52(1-3):53–100, 2004.

[BTS05] L. Birkedal and N. Torp-Smith. Higher order separation logic and abstraction. Submitted., February 2005.

[BTSY05] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 260–269, 2005.

[Cla01] D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.

[CN02] A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe*, volume 2391 of *LNCS*, pages 471–490, 2002.

[CNP01] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, pages 53–76, 2001.

[COB03] C. Calcagno, P. W. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Computer Science*, 298(3):557–581, 2003.

[dBP02] F.S. de Boer and C. Pierik. Computer-aided specification and verification of annotated object-oriented programs. In B. Jacobs and A. Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 163–177, 2002.

[DF01] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 59–69, 2001.

[DLN98] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research 156, DEC Systems Research Center, 1998.

[dRE98] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gri93] D. Gries. Data refinement and the tranform. In Manfred Broy, editor, *Program Design Calculi*. Springer, 1993. International Summer School at Marktoberdorf.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA '91 Conference Proceedings*, volume 26(11) of *SIGPLAN*. ACM, 1991.

[JKW03] B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 202–219, 2003.

[JLS04] B. Jacobs, K. R. M. Leino, and W. Schulte. Multithreaded object-oriented programs with invariants. In *SAVCBS*, 2004.

[Jon96] C. B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Kas06a] I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In *Formal Methods*, number 4085 in LNCS, pages 268–283, 2006.

[Kas06b] I. T. Kassios. A theory of object oriented refinement. PhD dissertation, University of Toronto, 2006.

[LCC+03] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. Springer, 2003.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[LM04] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 491–516, 2004.

[LM05] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *Proceedings, Formal Methods*, volume 3582 of *LNCS*, pages 26–42, 2005.

[LPHZ02] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 246–257, 2002.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2), 1995.

[LW94] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[Mey97] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[Mit86] J. C. Mitchell. Representation independence and data abstraction. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 263–276, 1986.

[MPHL04] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2004.

[MTSO04] I. Mijajlovic, N. Torp-Smith, and P. W. O'Hearn. Refinement and separation contexts. In *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, 2004.

[Mül02] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[MY05] I. Mijajlović and H. Yang. Data refinement with low-level pointer operations. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 19–36, 2005.

[Nau01a] D. A. Naumann. Ideal models for pointwise relational and state-free imperative programming. In Harald Sondergaard, editor, *ACM International Conference on Principles and Practice of Declarative Programming*, pages 4–15, 2001.

[Nau01b] D. A. Naumann. Patterns and lax lambda laws for relational and imperative programming. Technical Report 2001-2, Computer Science, Stevens Institute of Technology, 2001.

[Nau02] D. A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science*, 278(1–2):271–301, 2002.

[Nau05a] D. A. Naumann. Assertion-based encapsulation, object invariants and simulations. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Post-proceedings, Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *LNCS*, pages 251–273, 2005.

[Nau05b] D. A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, pages 190–204, 2005. Best Software Science Paper by the European Association of Software Sciences and Technology at the European Joint Conferences on Theory and Practice of Software (ETAPS) 2005.

[Nau06a] D. A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, number 4189 in LNCS, pages 279–296, 2006.

[Nau06b] D. A. Naumann. Observational purity and encapsulation. *Theoretical Computer Science*, 2006. To appear. Extended version of [Nau05b].

[NB04] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 313–323, 2004.

[NB06] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Computer Science*, 365:143–168, 2006. Extended version of [NB04].

[O'H05] P. W. O'Hearn. Scalable specification and reasoning: Technical challenges for program logic. In Bertrand Meyer and James C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, October 2005. Post-proceedings, to appear.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, 1995.

[OYR04] P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 268–280, 2004.

[Par05] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, November 2005. Dissertation.

[PCdB05] C. Pierik, D. Clarke, and F. S. de Boer. Controlling object allocation using creation guards. In *Proceedings, Formal Methods*, volume 3582 of *LNCS*, pages 59–74, 2005.

[PdB05a] C. Pierik and F.S. de Boer. On behavioral subtyping and completeness. In *ECOOP Workshop on Formal Techniques for Java-like Programs*. 2005. To appear.

[PdB05b] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Computer Science*, 2005. to appear.

[Pit97] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-Like Languages*, volume 2, chapter 17, pages 173–193. Birkhauser, 1997. Reprinted from *Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.

[Pit00] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[Plo73] G. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, University of Edinburgh, School of Artificial Intelligence, 1973.

[Rey81] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.

[Rey02] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[RM99] J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2–3):191–221, 1999.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, NY, second edition, 2002.

[SS05] C. Skalka and S. Smith. Static use-based object confinement. *Springer International Journal of Information Security*, 4(1-2), 2005.

[YO02] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *Proceedings, FOSSACS*, 2002.