Assertion-Based Encapsulation, Object Invariants and Simulations

David A. Naumann*

Department of Computer Science,
Stevens Institute of Technology, Hoboken, NJ 07030, USA

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. Encapsulation is assessed in terms of modular reasoning about invariants and simulations.

1 Introduction

This paper addresses two problems in reasoning about sequential object-oriented programs in languages like Java: reentrant callbacks and sharing of mutable objects. We present 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 difficulties have various manifestations in informal practice but can be understood most clearly in terms of formal reasoning, in particular 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. [5] and has been extended by Leino, Müller, and others [27, 7, 40, 43].

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. [36] and Jacobs et al. [24] give good introductions to the problems and other solution approaches. The book by Szyperski et al. [52] illustrates the problems 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 [22] may be helpful.

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 with simulations. Section 7 considers extension of the approach to invariants that depend on non-owned objects —a discipline for friendly cooperation. Section 8 discusses prospects and challenges for further extensions.

* Supported in part by the US National Science Foundation (CCR-0208984 and CCF-0429894) and by Microsoft.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 251–273, 2005.

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. 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.[1] 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 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*[2] 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 [23]. More precisely, it is not suitable to be visible to clients and is maintained solely by the methods of the object's class 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. As we shall explain, 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 pattern [22]. 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, because $notify$ is an *upcall* to the client. The difficulty is that $v.notify$ may make a *reentrant callback* to the sensor. 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* as control returns to s while another method invocation (here $update$) is in progress.

It is common that a reentrant callback is intended. In the example, $getval$ might simply read a field and cause no problem. However, trouble is likely if v invokes on s a

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

[2] 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".

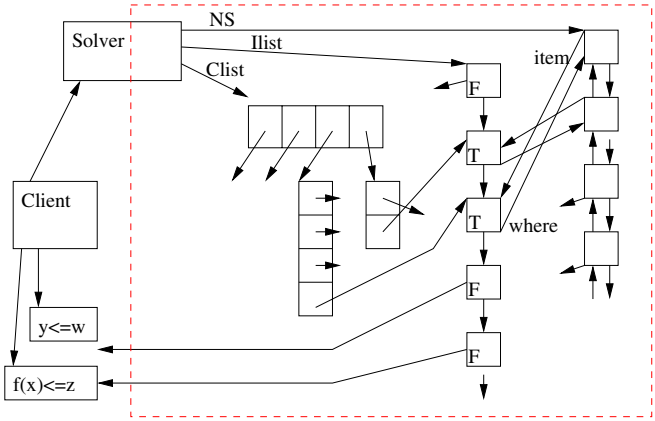


Fig. 1. An example data structure

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. In terms of the abstract interface, is v still in the set of registered views? In terms of the array, is v in fact in the array?

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 [49]. 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:[3]

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

The client program is intended to have a reference to the $Solver$ object, and 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

[3] 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 $\mathbf{self}.x$.)

be problematic because the client could update the object and falsify the invariant of $Solver$.

Suppose for simplicity that class $Solver$ has no proper subclasses or 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 other objects; 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.

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 from outsiders. This is the *dominator property* [16]: 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 callback. This could be done using static analysis for control flow, taking into account aliasing in order not to disallow too many programs. 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.

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 [29, 32] and is sometimes called *visible state semantics* [36]. 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 realistic 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 $z.f()$. This alternative must be rejected on grounds of information hiding, the predicate $\mathcal{I}^{Subject}$, like most invariants, depends on internals that should be encapsulated within the class.

Various techniques have been proposed to hide information about an invariant while expressing that it is in force. One alternative is to introduce a typestate [20] to stand for "the invariant is in force". Another approach is to treat its name as opaque with respect to its definition [8], as may be done in higher order logic using existential quantification [9]. 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 [26] 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 approach of Barnett et al. [5]. 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".[4] 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 $\mathbf{self}.inv$ 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 $\mathbf{self}.inv$.

```
class Subject {
  private x, y : int := 0, 1;
  private 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)

The discipline imposes several proof obligations in order to ensure that the following is a *program invariant*, i.e., it holds in all reachable states:

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

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 $\mathcal{I}$. 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:

- 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) upon truthification of $E.inv$. The first conjunct prevents reentrance to this region of code since **pack** sets $E.inv$ true.
- 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.

[4] Here $[o/\mathbf{self}]$ denotes substitution of o for $\mathbf{self}$, taking aliasing into account.

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 $notify$.

```
unpack self;
assert ¬self.inv
x := x + 1;
y := y + 1;
assert I^Subject
pack self;
view.notify(self);
```

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 in the code at right, where, as in (1), the assignment $y := y + 1$ precedes invocation of $notify$ so that the implementation of m can be verified. 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 code on the right. 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 $z.inv$.

```
unpack self;
assert ¬self.inv
x := x + 1;
view.notify(self);
y := y + 1;
assert I^Subject
pack self;
```

Summary. Through use of ghost field inv following the rules of the discipline, a harmful reentrant callback 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 $Subject$.

4 Sharing and Object Invariants

Let us set aside the issue of reentrance and consider another toy example, now involving shared references (see 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 $\mathcal{I}^{Subject2}[s/\mathbf{self}]$ 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 $s.inv \Rightarrow \mathcal{I}^{Subject2}[s/\mathbf{self}]$.

```
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

One diagnosis is that the invariant of $Subject2$ should not be allowed to depend on fields of objects other that **self**. Indeed, some proposals in the literature on invariants for object-oriented programs are only sound under this restriction [30]. But for many classes this is highly impractical. For an example, consider updates to the structure of the solver in Section 2, assuming list operations are coded in object-oriented style, e.g., using methods of the node classes for setting fields and for recursive list operations.

The name "$leak$" indicates our diagnosis: just as field x is private, so too the object referenced by x belongs within class $Subject2$. 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, or pass one as an argument to a client method.

The dominator property can be enforced using a type system such as the Universe system [35] and variations on Ownership Types [17, 13, 12, 1]. 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 name "$leak$" suggests that what has gone wrong in the example 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.[5]

Before turning to that topic we note that Separation Logic [50] provides a way to express that a predicate depends on only some objects in the heap (and correctness assertions that express on what part of the heap the correctness of a command depends). The logic has been used for encapsulating dependence of invariants in simple imperative programs [41] but in its current form the logic depends on a concrete view of heap cells in which all fields are explicit. This is at odds with subtyping and inheritance which affords modular reasoning about extensible classes. This is an exciting line of research, but adoption of a nonstandard logic for specification and verification has significant cost.

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. We define transitive ownership as a relation on references 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 $\mathbf{self} \succ p$.

In virtue of 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.[6]

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

[5] Skalka and Smith [51] also study use-based object confinement, for different purposes.

[6] 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.

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.

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. This means in a sense that control crosses 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:

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

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

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 $\neg o.inv$.

We have prevented interference not 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]

[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 via the **type** operator [26].

– Updates to ghost fields, including update of an unbounded number of objects (in **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 [45]. 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 [18]. 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 [5, 27]. A marked field declaration **rep** $f : T$ is syntactic sugar for the invariant $\mathbf{self}.f.own = \mathbf{self}$ and **peer** $f : T$ is syntactic sugar for $\mathbf{self}.f.own = \mathbf{self}.own$. It is also possible to infer, absent other annotation, that the implementation of a method with precondition $\mathbf{self}.inv$ should be annotated with **unpack self**; ... ; **pack self** so this need not be written explicitly.

The discipline supports an attractive extension to frame conditions: without mention in a modifies clause, a method can update committed objects. For details see [5].

Quantification. We have formalized the program invariants (2–4) using quantifications that range over all allocated objects. Quantification is problematic. If quantification ranges over currently allocated objects then a quantified formula can be falsified by garbage collection, e.g., $(\,\forall\, o \mid P(o)\,)$ is falsified if the only object with property P gets collected. This cannot happen if P connects o with other objects via ordinary fields, e.g., $P(o) = (o.f = C.x)$ with x a static field of class C, as then o is not garbage. But the problem occurs with as simple a property as $y.f = 1$ if y is a ghost field. Garbage sensitivity has been studied in depth by Calcagno et al. [14]. A workable solution 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 allocating a new object. Pierik, de Boer, and Clarke [43] have explored, for example, the Singleton pattern [22] 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. (Recall that our quantifications range over non-null object references; we write $\mathbf{type}(o)$ for the dynamic type of an object.)

This problem can be taken into account by including in the definition of admissibility that an invariant cannot be falsifiable by construction of new objects. The authors of the Boogie papers [5, 27] intend that invariants use quantification only over owned objects, which achieves this effect. Barnett and Naumann [40] 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. [43] based on a notion of update guard that we discuss 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

set-owner E **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 **set-owner** E **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.

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 [21] with the example of a lexer that owns an input stream but that stream is constructed 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 [41]. Other examples can be found in [27, 4].

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 $\mathbf{true}$.

Owned objects are now owned at a particular class, i.e., field own ranges over $\mathbf{null}$ 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 $\mathcal{I}^C$ depends.

The **pack** and **unpack** commands are revised to mention the class involved. For **unpack** E **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 **pack** E **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.

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

Methods are dynamically dispatched, which raises the question how to express the precondition that before was just $inv = \mathbf{true}$. The Boogie paper [5] introduces notation

```
class Subject2 { //Alternate version
  private rep x : Integer := new Integer(0);
  private rep z : int := 0;
  invariant I^Subject2'   where I^Subject2' =df 0 ≤ z
  method m() { x.incr(); }   }
```

Fig. 5. Revised $Subject2$

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 an inherited method.

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 [5] and more rigorously in [40]; see also [27]. Extension of the discipline to concurrent programs has also been investigated [25].

Completeness is another matter. It is not at all clear to this author how to formulate an interesting notion of completeness. Clearly it should 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? Related questions are which admissible predicates are expressible as formulas and 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 [44].

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 [28] and by Pierik et al. [43]. 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 Pointer Confinement and Simulation

Fig. 5 shows an alternate implementation of class $Subject2$ from Fig. 4. 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). 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.) The technique is practical because the simulation property only needs to be proved for the reimplemented methods: For arbitrary program contexts, simulation should follow from simulation for the revised class, by a general *representation independence* property of the language.

The technique was articulated by Hoare [23] drawing on work of Milner [33] and has seen much development for use with purely functional programs [48, 34, 47] as well as first order imperative and concurrent programs [31]. For first order imperative programs the topic is thoroughly surveyed in the textbook by de Roever et al. [19]. Object oriented programs have features in common with higher order imperative programs, for which representation independence is nontrivial owing to semantic difficulties [42, 46, 39]. 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 [15] —under the drastic simplification that copying is used instead of sharing. Their results have been used to validate laws of program refactoring [11, 10].

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 [2] are able to show 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 this 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 work of Banerjee and Naumann [2]. 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*.[9] 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.

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 [4]. 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, so 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. 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.

$$\begin{array}{l}\mathbf{class}\ \mathit{Main}\ \{\quad s : \mathit{Subject}2 := \mathbf{new}\ \mathit{Subject}2;\\ \quad \mathbf{method}\ \mathit{main}()\ \{\ i : \mathit{Integer} := s.\mathit{leak}();\ \mathit{Print}(i.\mathit{val});\ \}\quad \}\end{array}$$

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 [4] (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 regard it seems that the main advantage of the inv/own discipline over ownership types is the ability to temporarily violate the ownership property in order to transfer objects between owners. The Spec# project [6] is exploring inference to determine where the $\mathbf{set\text{-}owner}$, $\mathbf{pack}$, and $\mathbf{unpack}$ commands are needed. Integration with ownership types merits investigation.

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.

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

[9] 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.

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 [1] 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.

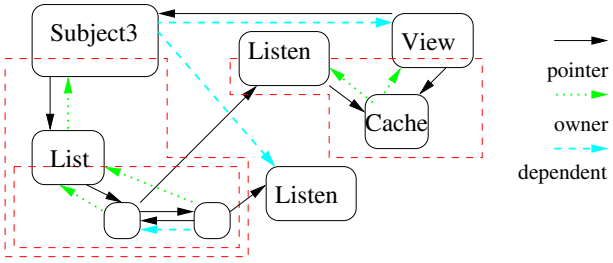
The need for object invariants to depend on non-owned objects arises in quite simple situations. In Section 2 we considered the $Solver$ invariant that involves doubly-linked list conditions $p.next = \mathbf{null} \vee p.next.prev = p$ for all p in $NS.next^*$. It is possible to associate the entire 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 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 —see the code in Fig. 6. 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

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

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



class *Subject*3 { val : **int**; vsn : **int**; $listeners : List\langle Listener\rangle$;
...}
class *Cache* { vsn : **int**; val : **int**; }
class *View* { $sbj : Subject$; **rep** $st : Cache$;
invariant $\mathcal{I}^{View}$ where $\mathcal{I}^{View} =_{df} sbj.vsn - 1 \leq st.vsn \leq sbj.vsn$
$\wedge (st.vsn = sbj.vsn \Rightarrow st.val = sbj.val)$;
...}
class *Listener* { $st : Cache$;
method *notify*() { ... } }

Fig. 6. Observer pattern using separate listeners

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 [35, 27, 36] 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. In 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 $v : View$ 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:

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

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?

Anyway, packing and unpacking is designed to embody hierarchical encapsulation. Here we are considering peers that are in some sense within the same encapsulation boundary. Moreover, to repack a view v, the $Subject$ would need to justify that $\mathcal{I}^{View}[v/\mathbf{self}]$ 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, so 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 [7] 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 "$\mathbf{friend}\ View\ \mathbf{reads}\ vsn, val$;" 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 $\mathbf{unpack}$ [7, 40].

The 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

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

protects the original invariant $\mathcal{I}^{View}$ including condition $sbj.vsn - 1 \leq st.vsn$. This is because the proof obligation imposed on $View$ for U is satisfied:

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

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

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

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 [30]. 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 [30]. 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.

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 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 [7].

8 Challenges for Future Work

We have not exhausted the issues brought up by the last example. 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 increasingly elaborate patterns have motivated increasingly complicated ownership type systems and may well necessitate more complicated versions of the inv/own and friendship disciplines. We mention two more ways in which the friendship discipline, as currently formulated [7, 40], is inadequate. Consider a variation on $Subject3$ where its state is not just val : **int** but rather some data structure; then $\mathcal{I}^{View}$ would depend not on $sbj.val$, but on $sbj.f.g\ldots$, i.e., a path 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 and it appears nontrivial to generalize the conditions to longer paths owing to the various possibilities of sharing.

The second inadequacy of the current friendship discipline is that each atomic update must restore the friend's invariant. In the case at hand, the friend's invariant depends on two fields val and vsn. It happens that if $sbj.val$ is updated before $sbj.vsn$ then the discipline can be followed, but in general one would like to require the invariant to hold only after several related updates are done. Perhaps this can be achieved by a protocol with a ghost variable to track whether the friendship dependence is in effect? Would that explicit expression of atomicity lead to very different reasoning than using history constraints?

While incremental extensions can be made to address these two inadequacies 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 con-

figuration, expressed with the help of ghost fields to encode the configuration[10] 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.

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?"[11] Another possible source of inspiration is Separation Logic, which offers notation that can transparently depict groups of objects and their interrelation. 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 has been given a semantic foundation [37, 38] but not thoroughly investigated. 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 [41]. 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. Thanks to the organizers of FMCO 2004 for the opportunity to present this work and meet with other researchers in such a congenial and supportive environment. This paper reflects feedback from the meeting as well as corrections and suggestions from reviewers and from Mike Barnett.

References

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

[2] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, revision pending. Extended version of [3].

[3] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 166–177, 2002.

[4] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005. To appear.

[5] 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.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS post-proceedings*, 2004.

[7] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction*, pages 54–84, 2004.

[10] The Aldrich-Chambers system might help here [1].

[11] Personal communication, April 2004.

[8] G. Bierman and M. Parkinson. Separation logic and abstraction. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 247–258, 2005.
[9] L. Birkedal and N. Torp-Smith. Higher order separation logic and abstraction. Submitted., Feb. 2005.
[10] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Programming*, 52(1-3):53–100, 2004.
[11] 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.
[12] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
[13] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 213–223, 2003.
[14] C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Comput. Sci.*, 298(3):557–581, 2003.
[15] 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.
[16] D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.
[17] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, 2001.
[18] F. 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.
[19] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
[20] 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.
[21] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research 156, DEC Systems Research Center, 1998.
[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[23] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
[24] 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)*, LNCS.
[25] B. Jacobs, K. R. M. Leino, and W. Schulte. Multithreaded object-oriented programs with invariants. In *SAVCBS*, 2004.
[26] 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 F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. 2003.
[27] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 491–516, 2004.
[28] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *Formal Methods*, 2005.
[29] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
[30] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6), 1994.

[31] N. Lynch and F. Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2), 1995.
[32] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.
[33] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.
[34] J. C. Mitchell. Representation independence and data abstraction. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 263–276, 1986.
[35] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
[36] 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.
[37] D. A. Naumann. Ideal models for pointwise relational and state-free imperative programming. In H. Sondergaard, editor, *ACM International Conference on Principles and Practice of Declarative Programming*, pages 4–15, 2001.
[38] D. A. Naumann. Patterns and lax lambda laws for relational and imperative programming. Technical Report 2001-2, Computer Science, Stevens Institute of Technology, 2001.
[39] D. A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Comput. Sci.*, 278(1–2):271–301, 2002.
[40] 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.
[41] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 268–280, 2004.
[42] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, 1995.
[43] C. Pierik, D. Clarke, and F. S. de Boer. Controlling object allocation using creation guards. In *Formal Methods 2005*, 2005.
[44] C. Pierik and F. de Boer. On behavioral subtyping and completeness. In *ECOOP Workshop on Formal Techniques for Java-like Programs*. 2005. To appear.
[45] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Comput. Sci.*, 2005. to appear.
[46] 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.
[47] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.
[48] G. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, University of Edinburgh, School of Artificial Intelligence, 1973.
[49] Rehof and Mogensen. Tractable constraints in finite semilattices. *Sci. Comput. Programming*, 1996.
[50] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
[51] C. Skalka and S. Smith. Static use-based object confinement. *Springer International Journal of Information Security*, 4(1-2), 2005.
[52] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, NY, second edition, 2002.