Friends Need a Bit More: Maintaining Invariants Over Shared State

February 6, 2004

Mike Barnett and David A. Naumann*

[0] Microsoft Research
mbarnett@microsoft.com
[1] Stevens Institute of Technology
naumann@cs.stevens-tech.edu

Abstract. A *friendship system* is introduced for modular static verification of object invariants. It extends a previous methodology, based on ownership hierarchy encoded in auxiliary state, to allow for state dependence across ownership boundaries. Friendship describes a formal protocol for a *granting class* to grant a *friend class* permission to express its invariant over fields in the granting class. The protocol permits the safe update of the granter's fields without violating the friend's invariant. The ensuing proof obligations are minimal and permit many common programming patterns. A soundness proof is sketched. The method is demonstrated on several realistic examples, showing that it significantly expands the domain of programs amenable to static verification.

0 Introduction

Whether they are implicit or explicit, object invariants are an important part of object-oriented programming. An object's invariant is, in general, a *healthiness* guarantee that the object is in a "good" state, i.e., a valid state for calling methods on it.

For example, in a base class library for collection types, certain method calls may be made on an enumerator only if the underlying collection has not been modified since the enumerator was created. Other examples are that an acyclic graph is indeed acyclic or that a sorted array has its elements in the proper order.

Various proposals have been made on how object invariants can be formally expressed and on different mechanisms for either guaranteeing that such invariants hold [LN02,LG86,Mül02] or at least dynamically recognizing moments in execution where they fail to hold [BS03,CL02,Mey97]. For the most part, these systems require some kind of partitioning of heap objects so that an object's invariant depends only on those objects over which it has direct control. This is intuitive, since it is risky for one data structure to depend on another over which it has no control. However, systems such as *ownership types* [CNP01,Cla01,BLS03,Mül02] are inflexible in that they demand object graphs to be hierarchically partitionable so that the dependencies induced by object invariants do not cross ownership boundaries. There are many situations where an object depends on another object but cannot reasonably own it.

* Partially supported by NSF CCR-0208984; SRI Visiting Fellowship; Microsoft Research.

We relax these restrictions with a new methodology; our method allows the definition of a protocol by which a *granting class* can give priviliges to another *friend class* that allows the invariant in the friend class to depend on fields in the granting class. As in real life, friendship demands the active cooperation of both parties. A friend class can publish restrictions on field updates of the granting class. The granting class must be willing to operate within these restrictions. In return, each instance of the friend class must register itself with the instance of the granting class that it is dependent on. And as in real life, the quality of the friendship depends on how onerous its burdens are. We believe our system imposes a minimal set of constraints on the participating parties.

Our method builds on ownership-based invariants [BDF+03a], formalized using an auxiliary field $owner$ [LM03]. We refer to the combination of [BDF+03a] and [LM03] as the *Boogie methodology*. An on-going project at Microsoft Research named "Boogie" is building a tool based on that methodology. In Section 1, we review the relevant features of the object invariant system from that work.

Section 2 presents a representative example of an instance of a granting class performing a field update that could violate the invariant of an instance of a friend class. We describe the required proof obligations for the granting object to perform the field update without violating the invariants of its friends or the object invariant system. In Section 2.0, we describe how a granting class declares which classes are its friends and how granting objects track friends that are dependent upon it. It is important that a granting object has an abstraction of the invariants of its friends, rather than the full details. Our method for this is explained in Section 2.1. Then in Section 2.2, we define the obligations incumbent upon the friend class for notifying granting objects of the dependence. Section 2.3 summarizes all of the features of our method.

Section 3 sketches a soundness argument for our method. Section 4 describes two extensions. The first, in Section 4.0, presents a convenient methodology that shows how reasoning about dependents can be linked to the code of the granting class. In Section 4.1, we describe a syntactic means for transmitting information after a field update back to the granting object from a friend object. We give several examples in Section 5 that present our methodology in action on challenging examples. Section 6 reviews related work and Section 7 summarizes our contribution and points out future work.

We assume some familiarity with the principles of object-oriented programming and the basics of assertions (pre- and post-conditions, modifies clause, and invariants) as well as their use in the static modular verification of sequential object-oriented programs. However, we do not presuppose any particular verification technology.

For simplicity, we describe our method without taking into account subtyping. The full treatment is described in a forthcoming technical report; it follows the same pattern as the Boogie work [BDF+03a]. A companion paper [NB04] gives a rigorous proof of soundness in a semantic model. The concrete syntax that we use is not definitive and illustrates one particular way to encode the information needed by our method.

```
class Set {
  fst : Node := null;
  insert(x : int)
  {
    t : Node := new Node(x);
    "code to insert t"
  }
  remove(x : int)
  {"delete first node with val x" }
  map(g : Fun)
  {"apply g to all elements; remove duplicates"}
}

class Node {
  val : int
  next : Node
  Node(x : int)
  {val := x; next := null}
}
class Fun {
  apply(x : int) : int
  { return x mod 7; }
}
```

Fig. 0. A set of integers is represented by a linked list, without duplicate values, rooted at fst. Method $insert$ adds an element if not already present. Method $map(g)$ updates the set to be its image through $g.apply$. Class $Node$ has only a constructor; nodes are manipulated in Set.

1 Using auxiliary fields for ownership-based invariants: A Review

Using the contrived example code in Figure 0 we review the Boogie approach to invariants and ownership. In our illustrative object-oriented language, class types are implicitly references; we use the term "object" to mean object reference.

An instance of class Set maintains an integer set represented by a sequence without duplicates, so that $remove(x)$ can be implemented by a linear search that terminates as soon as x is found. The specification of class Set could include invariant

Inv_{Set} : fst is the root of an acyclic sequence without duplicate values.

We denote the invariant for a class T by Inv_T. Note that since the invariant mentions instance fields, it is parameterized by an instance of type T. We write $Inv_T(o)$ where o is an object of type T when we want to make explicit the value of an invariant for a particular instance.

An object invariant is typically conceived as a pre- and post-condition for every method of the class. For example, if $remove(x)$ is invoked in a state where there are duplicates, it may fail to establish the intended postcondition that x is not in the set. Constructors establish invariants.

The method map takes the function supplied as an argument and, abstractly, maps the function over the set to yield an updated set. Suppose it is implemented by first updating all of the values in place and only after that removing duplicates to re-establish the invariant. One difficulty in maintaining object invariants is the possibility of reentrant calls: If an object g has access to the instance s on which $s.map(g)$ is invoked, then within the resulting call to $g.apply$ there could be an invocation of $s.remove$. But s at that point is in an inconsistent state —i.e., a state in which $Inv_{Set}(s)$ does not hold. It is true that by considering the body of $apply$ as given in Figure 0 we can rule out this

possibility. But for modular reasoning about Set we would only have a specification for $apply$ —quite likely an incomplete one. (Also, Fun could have been written as an interface; or $apply$ can be overridden in a subclass of Fun.)

A sound way to prevent the problem of re-entrance is for the invariant to be an *explicit* precondition and postcondition for every method: $apply$ must establish $Inv_{Set}(s)$ before invoking $s.remove$, and it cannot do so in our scenario. But this solution violates the principle of information hiding: Using $Node$ and maintaining Inv_{Set} are both decisions that might be changed in a revision of Set (or in a subclass). Indeed, we might want the field fst to be private to Set whereas the precondition of a public method should mention only visible fields.

It is possible to maintain proper encapsulation by making it the responsibility of Set to ensure that its invariant hold at *every* "observable state", not only at the beginning and end of every method but also before any "out" call is made from within a method. In the example, Set would have to establish $Inv_{Set}(s)$ within map before each call to $apply$. Though frequently proposed, this solution is overly restrictive. For instance, it would disallow the sketched implementation of map in which removal of duplicates is performed only after all the calls to $apply$. In a well structured program with hierarchical abstractions there are many calls "out" from an encapsulation unit, most of which do not lead to reentrant callbacks.

Programmers often guard against reentrant calls using a "call in progress" field; this field can be explicitly mentioned in method specifications. In some respects this is similar to a lock bit for mutual exclusion in a concurrent setting. Disallowing a call to $remove$ while a call to map is in progress can be seen as a protocol and it can be useful to specify allowed sequences of method invocations [DF01,DF03].

We wish to allow reentrant calls. They are useful, for example, in the ubiquitous Subject-View pattern where a reentrant callback is used by a View to inspect the state of its Subject. On the other hand, general machinery for call protocols seems onerous for dealing with object invariants in sequential programs. Moreover this is complicated by subclassing: a method added in a subclass has no superclass specification to be held to.

Boogie associates a boolean field inv with the object invariant. This association is realized in the following *system invariant*, a condition that holds in *every* state. (That is, at every control point in the program text.)

$$(\,\forall\, o \bullet\ o.inv\ \Rightarrow\ Inv_T(o) \quad \text{where}\ T = \mathbf{type}(o)\ \,) \tag{0}$$

Here and throughout the paper, quantification ranges over objects allocated in the current state. The dynamic (allocated) class of o is written $\mathbf{type}(o)$.

As part of the methodology to ensure that (0) is in fact a system invariant, we stipulate that the auxiliary field inv may only be used in specifications and in special statements **pack** and **unpack**. If the methods of Set all require inv as precondition, then $apply$ is prevented from invoking $s.remove$ as in the first solution above —but without exposing Inv_{Set} in a public precondition. Nevertheless, the body of $remove$ can be verified under precondition Inv_{Set} owing to precondition inv and system invariant (0).

The special statements **pack** and **unpack** enforce a discipline to ensure that (0) holds in every state. Packing an object sets inv to **true**; it requires that the ob-

ject's invariant holds. Unpacking an object sets inv to $\mathbf{false}$. Since an update to the field $o.f$ could falsify the invariant of o, we require that each update be preceded by $\mathbf{assert}\ \neg o.inv$.

The details are deferred so we can turn attention to another issue raised by the example, namely representation exposure. The nodes reached from $Set.fst$ are intended to comprise an encapsulated data structure, but even if fst is declared private there is a risk that node references are *leaked*: e.g., some client of a set s could change the value in a node and thereby falsify the invariant. Representation exposure due to shared objects has received considerable attention [LN02], including ownership type systems [Mül02,CD02,BN02a,BLS03] and Separation Logic [OYR04]. In large part these works are motivated by a notion of *ownership*: the $Nodes$ reached from $s.fst$, on which $Inv_{Set}(s)$ depends, are owned by that instance s and should not be accessed except by s. This ensures that the invariant of s is maintained so long as methods of Set maintain it.

The cited works suffer from inflexibility due to the conservatism necessary for static enforcement of alias confinement. For example, type systems have difficulty with transfering ownership. However, transfer is necessary in many real-world examples and state encapsulation does not necessarily entail a fixed ownership relation. (This is emphasized in [OYR04,BN03].)

A more flexible representation of ownership can be achieved using auxiliary fields $owner$ and $comm$ in the way proposed by Barnett *et al.* and refined by Leino and Müller [LM03]. The field $owner$, of type $Object$, designates the owner, or $\mathbf{null}$ if there is no owner. The boolean field $comm$ designates whether the object is currently *committed to* its owner: if it is true, then its invariant holds and its owner is depending on having sole access for modifying it. The latter is true whenever the owner, o, sets its own inv bit, $o.inv$. Since o's invariant may depend on the objects that it owns, it cannot guarantee its invariant unless no other object can update any object p where $p.owner = o$, or where p is a transitively owned object. There are two associated system invariants. The first is that $o.inv$ implies that every object p owned by o is committed.

$$(\forall\, o \bullet\ o.inv\ \Rightarrow\ (\forall\, p \bullet\ p.owner = o\ \Rightarrow\ p.comm\ )\ ) \tag{1}$$

The second ties committment to invariants:

$$(\forall\, o \bullet\ o.comm\ \Rightarrow\ o.inv\ ) \tag{2}$$

The special fields $inv, comm, owner$ are allowed in pre- and post-conditions; only $owner$ is allowed to occur in object invariants. A consequence is that in a state where o transitively owns p, we have $o.inv\ \Rightarrow\ p.comm$.

The point of ownership is to constrain the dependence of invariants and to encapsulate the objects on which an invariant Inv_T depends so that it cannot be falsified except by methods of T.

Definition 1 (admissible object invariant). *An* admissible object invariant $Inv_T(o)$ *is one such that in any state, if* $Inv_T(o)$ *depends on some object field* $p.f$ *in the sense that update of* $p.f$ *can falsify* $Inv_T(o)$*, then either*

- $p = o$ *(this means that* $\mathbf{this}.f$ *is in the formula for* Inv_T*); or*
- p *is transitively owned by* o.

Transitive ownership is inductively defined to mean that either $p.owner = o$ or that $p.owner$ is transitively owned by o.

Remarkably, f is allowed to be public, though for information hiding it is often best for it to be private or protected. The ownership discipline makes it impossible for an object to update a public field of another object in a way that violates invariants.

Aside 1 *The methodology handles situations where an object owns others that it does not directly reference, e.g., nodes in a linked list. But a common situation is direct reference like field* fst*. To cater for this, it is possible to introduce a syntactic marker* $\mathbf{rep}$ *on a field, to designate that its value is owned. It is not difficult to devise annotation rules to maintain the associated system invariant*

$$(\forall\, o : T \bullet\ o.inv\ \wedge\ o.f \neq \mathbf{null}\ \Rightarrow\ o.f.owner = o\ )$$

for every $\mathbf{rep}$ *field* f *declared in each class* T*. On the other hand, one can simply include "*$\mathbf{this}.f\ =\ \mathbf{null}\ \vee\ \mathbf{this}.f.owner\ =\ o$*" as a conjunct of the invariant, so in this paper we omit this feature. A similar feature is to mark a field* f *as* $\mathbf{peer}$*, to maintain the invariant* $\mathbf{this}.f = \mathbf{null} \vee \mathbf{this}.f.owner = \mathbf{this}.owner$ *[LM03]. Again, it is useful but does not solve the problems addressed in this paper and is subsumed under our proposal.*

The system invariants hold in every state —loosely put, "at every semicolon"— provided that field updates to the field f, with expressions E and D, are annotated as

$$\begin{array}{l} \mathbf{assert}\ \neg E.inv; \\ E.f := D; \end{array} \tag{3}$$

and the special fields inv, $comm$, and $owner$ are updated only by the special statements defined below. Most important are the special statements for inv and $comm$.[0]

$$\begin{array}{lcl} \mathbf{unpack}\ E & \equiv & \mathbf{assert}\ E \neq \mathbf{null} \wedge E.inv \wedge \neg E.comm; \\ & & E.inv := \mathbf{false}; \\ & & \mathbf{foreach}\ p\ \mathbf{such\ that}\ p.owner = E\ \mathbf{do}\ p.comm := \mathbf{false}; \end{array}$$

$$\begin{array}{lcl} \mathbf{pack}\ E & \equiv & \mathbf{assert}\ E \neq \mathbf{null} \wedge \neg E.inv \wedge Inv_T(E) \\ & & \quad \wedge\ (\forall\, p \bullet\ p.owner = E\ \Rightarrow\ \neg p.comm \wedge p.inv\ ); \\ & & \mathbf{foreach}\ p\ \mathbf{such\ that}\ p.owner = E\ \mathbf{do}\ p.comm := \mathbf{true}; \\ & & E.inv := \mathbf{true}; \end{array}$$

Proofs that $\mathbf{pack}$ and $\mathbf{unpack}$ maintain the system invariants (0), (1), and (2) can be found in [BDF$^+$03a] and [NB04]. Let us consider how (3) maintains (0). An admissible

[0] Note that the "$\mathbf{foreach}$" statement in $\mathbf{pack}$ updates the auxiliary field $comm$ of an unbounded number of objects. An equivalent expression, more in the flavor of a specification statement in which the field $comm$ is viewed as an array indexed by objects, is this: $\mathbf{change}\ comm\ \mathbf{such\ that}\ (\forall\, p \bullet\ p.comm\ \equiv\ p.comm_0\ \wedge\ p.owner \neq E\ )$.

invariant for an object o depends only on objects owned by o and thus can only be falsified by update of the field of such an object. But an update of $p.f$ is only allowed if $\neg p.inv$. If p is owned by o then $\neg p.inv$ can only be achieved by unpacking p, which can only be done if p is not committed. But to un-commit p requires unpacking o —and then, since $\neg o.inv$, there is no requirement for $Inv_T(o)$ to hold.

The special statements **pack** and **unpack** effectively impose a hierarchical discipline of ownership, consistent with the dependence of invariants on transitively owned objects. Because the discipline is imposed in terms of auxiliary state and verification conditions rather than as an invariant enforced by a static typing system [Mül02,Cla01,BLS03,BN02a], the temporary violations permitted by **pack** and **unpack** offer great flexibility.

Every constructor begins implicitly with initialization

$$inv, comm, owner := \mathbf{false}, \mathbf{false}, \mathbf{null}.$$

The last of the special statements is used to update $owner$.

$$\begin{array}{l}\textbf{set-owner } E \textbf{ to } D \ \equiv \\ \quad \textbf{assert } E \neq \mathbf{null} \wedge \neg E.inv \wedge (D = \mathbf{null} \vee \neg D.inv); \\ \quad E.owner := D;\end{array}$$

At first glance it might appear that the precondition $E.owner = \mathbf{null} \vee \neg E.owner.inv$ is needed as well, but for non-null $E.owner$, we get $\neg E.owner.inv$ from $\neg E.inv$ by the system invariants.

A cycle of ownership can be made using **set-owner**, but the precondition for **pack** cannot be established for an object in such a cycle.

One of the strengths of this approach to ownership is that **set-owner** can be used to transfer ownership as well as to initialize it (see the example in Section 5.2). Another strength is the way invariants may be declared at every level of an inheritance chain; we have simplified those few parts of the methodology which are concerned with subclassing. The reader may refer to the previous papers [BDF+03a,LM03] for more discussion.

2 The problem: objects without borders

The Boogie methodology is adequate for the maintenance of ownership-based invariants. Our contribution in this paper, summarized in Section 2.3, is to go beyond ownership to cooperating objects.

We describe the problem and our method using the code in Fig. 1. The invariant $0 \leq time$ in class $Master$ abbreviates $0 \leq \mathbf{this}.time$. (Recall that $Inv_{Master}(o)$ denotes $0 \leq o.time$.) According to the rules for admissible invariants in Section 1, Inv_{Master} is allowed.

The constructor for $Master$ exemplifies the usual pattern for constructors: it first initializes the fields in order to establish the invariant and then uses **pack** to set the inv bit. Methods that update state typically first execute **unpack** to turn off the inv bit and then are free to modify field values. Before they return, they use **pack** once their invariant has been reestablished.

```
class Master {
  time : int;
  invariant 0 ≤ time;
  Master()
    ensures inv ∧ ¬comm;
  { time := 0; pack this; }
  Tick(n : int)
    requires inv ∧ ¬comm ∧ 0 ≤ n;
    modifies time;
    ensures time ≥ old(time);
  {
    unpack this;
    time := time + n;
    pack this;
  }
}
```

```
class Clock {
  t : int;
  m : Master;
  invariant m ≠ null ∧ 0 ≤ t ≤ m.time;
  Clock(mast : Master)
    requires mast ≠ null ∧ mast.inv;
    ensures inv ∧ ¬comm;
  { m := mast; t := 0; pack this; }
  Sync()
    requires inv ∧ ¬comm;
    modifies t;
    ensures t = m.time;
  { unpack this; t := m.time; pack this; }
}
```

Fig. 1. A simple system for clocks synchronized with a master clock. $Inv_{Clock}(\mathbf{this})$ depends on $\mathbf{this}.m.time$ but does not own $\mathbf{this}.m$.

The predicate Inv_{Clock} is not an admissible invariant: it depends on $m.time$, but a clock does not own its master. Otherwise a master could not be associated with more than one clock. While it might be reasonable to let the master own the clocks that point to it, we wish to address situations where this ownership relation would not be suitable. More to the point, such a solution would only allow Inv_{Master} to depend on the clocks whereas we want Inv_{Clock} to depend on the master.

Although Inv_{Clock} is not admissible according to Definition 1, the update of $time$ in $Tick$ increases the value of $time$, which cannot falsify Inv_{Clock}. The problem that our methodology solves is to allow non-ownership dependence in a situation like this, i.e., to support modular reasoning about the cooperative relationship wherein $Tick$ does not violate Inv_{Clock}.

However, while $Tick$ is a safe method in relation to Inv_{Clock}, we want to preclude the class $Master$ from defining a method $Reset$:

```
Master.Reset()
  requires inv;
  modifies time;
{ unpack this; time = 0; pack this; }
```

This is easily shown correct in terms of Inv_{Master}, but $o.Reset$ can falsify the invariant of any clock c with $c.m = o$. If we allow Inv_{Clock} to depend on $m.time$ and yet prevent this error, a precondition stronger than that in (3) must be used for field update. (In Section 4.0, we show how $Reset$ can be correctly programmed without violating the invariant of $Clock$.)

Leino and Müller's discipline [LM03], strengthens (3) to yield the following annotation:

$$\begin{array}{l}\mathbf{assert}\ \neg\mathbf{this}.inv\ \wedge\ (\,\forall\, p\ \mid\ \mathbf{type}(p) = \mathit{Clock}\ \bullet\ \neg p.inv\ );\\ \mathbf{this}.time := 0;\end{array}$$

Unfortunately, this does not seem to be a very practical solution. How can modular specifications and reasoning about an arbitrary instance of $Master$ hope to establish a predicate concerning all clocks whatsoever, even in the unlikely event that the predicate is true? Given the ownership system, it is also unlikely that an instance of $Master$ would be able to **unpack** any clock that refers to it via its m field and whose inv field was true.

Consider taking what appears to be a step backwards, concerning the Boogie methodology. We could weaken the annotation in the preceding paragraph to allow the master to perform the field update to $time$ as long as it does not invalidate the invariants of any clocks that could possibly be referring to it.

$$\begin{array}{l}\mathbf{assert}\ \neg\mathbf{this}.inv\ \wedge\\ \qquad (\,\forall\, p\ \mid\ \mathbf{type}(p) = \mathit{Clock}\ \bullet\ \neg p.inv\ \vee\ (\mathit{Inv}_{\mathit{Clock}}(p))_0^{\mathbf{this}.time}\ );\\ \mathbf{this}.time := 0;\end{array}$$

The substitution expression P_E^x represents the expression P with all unbound occurrences of x replaced by E, with renaming as necessary to prevent name capture. We use substitution to express the weakest precondition.[1] But the revised precondition does not appear to provide any benefit: while $\neg\mathbf{this}.inv$ is established by the preceding **unpack** in $Reset$, there is still no clear way to establish either of the disjuncts for arbitrary instances of $Clock$. In addition, as stated, this proposal has the flaw that it exposes Inv_{Clock} outside of class $Clock$.

We solve both of these problems. Given the following general scheme:

$$\begin{array}{l}\mathbf{assert}\ \neg E.inv\ \wedge\ (\,\forall\, p, T\ \mid\ \ldots\ \bullet\ \neg p.inv\ \vee\ (\mathit{Inv}_T(p))_D^{E.f}\ );\\ E.f := D;\end{array} \tag{4}$$

where the missing condition "$\ldots$" somehow expresses that $\mathbf{type}(p) = T$ and $Inv_T(p)$ depends on E, our methodology provides a way to manage the range of p and a way to abstract from $(Inv_T(p))_D^{E.f}$.

In the following three subsections we first deal with restricting the range of p in (4). Then we show how to abstract from $(Inv_T(p))_D^{E.f}$ in (4) to achieve class-oriented information hiding. Finally we complete the story about the range of p and redefine admissible invariants.

2.0 Representing Dependence

The first problem is to determine which objects p have $Inv_{Clock}(p)$ dependent on a given instance of $Master$. (In general, there could be other classes with invariants that

[1] Substitution for updates of object fields can be formalized in a number of ways and the technical details are not germane in this paper [AO97,FLL+02]. In general, object update has a global effect, and our aim is to achieve sound localized reasoning about such updates.

depend on instances of $Master$, further extending the range of p needed for soundness.) To allow for intentional cooperation, we introduce an explicit *friend declaration*

$$\textbf{friend}\ Clock\ \textbf{reads}\ time;$$

in class $Master$.[2] For a friend declaration appearing in class T':

$$\textbf{friend}\ T\ \textbf{reads}\ f;$$

we say T' is the *granting* class and T the *friend*. Field f is visible in code and specifications in class T. (Read access is sufficient.) There are some technical restrictions on f listed in Section 2.3. If in fact $Inv_T(p)$ depends on $o.f$ for some granting object o then o is reachable from p. For simplicity in this paper, we confine attention to paths of length one, so $o = p.g$ for some field g which we call a *pivot field*. (We also allow $p.g.f.h$ in $Inv_T(p)$, where h is an immutable field of f, e.g., the length of an array.)

One of the benefits of our methodology is to facilitate the decentralized formulation of invariants which lessens the need for paths in invariants. An example is the condition linking adjacent nodes in a doubly-linked list: reachability is needed if this is an invariant of the list header, but our methodology allows us to maintain the invariant by imposing a local invariant on each node that refers only to its successor node; see the example in Section 5.2.

To further restrict the range of p in (4), to relevant friends, we could explore more complicated syntactic conditions, but with predictable limitations due to static analysis. We choose instead to use auxiliary state to track which friend instances are susceptible to having their invariants falsified by update of fields of a granting object.

We introduce an auxiliary field $deps$ of type "set of object". We will arrange that for any o in any state, $o.deps$ contains all p such that $p.g = o$ for some pivot field g by which $Inv(p)$ depends as friend on some field of o. As with $owner$, this facilitates making explicit the relevant program and system invariants. Both $owner$ and $deps$ function as "back pointers" in the opposite direction of a dependence.

The associated system invariant is roughly this:

$$(\forall\, o : T' \bullet\ (\forall\, p : T \mid\ p.inv\ \wedge\ \text{“}Inv_T(p) \text{ depends on } o.f\text{”}\ \bullet\ p \in o.deps\ )) \quad (5)$$

for every T, T' such that T is a friend of T' reading f. That dependence happens via a pivot field will become clear later when we define admissibility for invariants.

We have reached the penultimate version of the rule for update of a field with friend dependents:

$$\begin{array}{l}\textbf{assert}\ \neg E.inv\ \wedge\ (\forall\, p \mid\ p \in E.deps\ \bullet\ \neg p.inv\ \vee\ (Inv_{\textbf{type}(p)}(p))^{E.f}_{D}\ );\\ E.f := D;\end{array} \quad (6)$$

A friend declaration could trigger a requirement that field updates in the granting class be guarded as in (6) and one could argue that in return for visibility of f in T, Inv_T should simply be visible in T'. This is essentially to say that the two classes are in a single module. Our methodology facilitates more hiding of information than that, while allowing cooperation and dealing with the problem of the range of p in (6). In the next subsection we eliminate the exposure of Inv in this rule, and then in the following subsection we deal with reasoning about $deps$.

[2] Similar features are found in languages including C++ and C#, and in the Leino-Müller work.

2.1 Abstracting from the friend's invariant

Our solution is to abstract from $(Inv_T)^{E.f}_D$ not as an auxiliary field but as a predicate U (for *update guard*). The predicate U is declared in class T, and there it gives rise to a proof obligation, roughly this: if both the friend object's invariant holds and the update guard holds, then the assignment statement will not violate the friend object's invariant. This predicate plays a role in the interface specification of class T, describing not an operation provided by T but rather the effect on T of operations elsewhere. There is a resemblance to behavioral assumptions in Rely-Guarantee reasoning for concurrent programs [Jon83,dRdBH+01].

In the friend class T it is the pivot field g and the friend field f that are visible, not the expressions E and D in an update that occurs in the code of the granting class T'. So, in order to define the update guard we introduce a special variable $\mathbf{val}$ to represent the value the field is being assigned:

$$\mathbf{guard}\ g.f := \mathbf{val}\ \mathbf{by}\ U(\mathbf{this}, g, val);$$

This construct appears in the friend class and must be expressed in terms that are visible to the granting class (thus allowing the friend class to hide its private information). We write $U(friend, granter, val)$ to make the parameters explicit. That is, U is defined in the context of T using vocabulary $(\mathbf{this}, g, \mathbf{val})$ but instantiated by the triple (p, E, D) at the update site in a granter method (see (6) and below). For example, the update guard declared in the friend class $Clock$ is:

$$\mathbf{guard}\ m.time := \mathbf{val}\ \mathbf{by}\ m.time \leq \mathbf{val};$$

Thus $U_{Clock}(\mathbf{this}, m, \mathbf{val}) \equiv m.time \leq \mathbf{val}$. Notice that $\mathbf{this}$ does not appear in this particular update guard. That is because, as stated earlier, it does not depend on the state of the instance of $Clock$.

Like a method declaration, an update guard declaration imposes a proof obligation. The obligations on the friend class T are:

$$Inv_T(\mathbf{this}) \ \wedge\ U(\mathbf{this}, g, \mathbf{val}) \ \Rightarrow\ (Inv_T(\mathbf{this}))^{g.f}_{\mathbf{val}} \tag{7}$$

for each pivot g of type T' and friend field f. A suitable default is to take U to be $\mathbf{false}$ so that the proof obligation is vacuous. Then the update rule is equivalent to that in [LM03]. At the other extreme, if, despite the declarations, Inv does not in fact depend on the pivot then U can be taken to be $\mathbf{true}$.

We have now reached the final version of the rule for update of a friend field:

$$\begin{array}{l}\mathbf{assert}\ \neg E.inv \wedge (\forall p \mid p \in E.deps \bullet \neg p.inv \vee U(p, E, D));\\ E.f := D;\end{array} \tag{8}$$

We are now in a position that a field update may be performed without violating the invariants of an object's friends by establishing the precondition

$$(\forall p \mid p \in E.deps \bullet \neg p.inv \vee U(p, E, D))$$

where U was written by the author of the class T in such a way that the class T' is able to (at least potentially) satisfy it. That is, it is an expression containing values and variables that are accessible in the context of T' and need not involve the private implementation details of T.

In the design of class T, some state variables may be introduced and made visible to T precisely in order to express U, without revealing too much of the internal representation of T. We pursue this further in Section 4.1.

For the clock example, $U_{Clock}(p, \mathbf{this}, time + n) = time \leq time + n$ which follows easily from precondition $0 \leq n$ of method $Tick$; thus the update precondition can be established independent from any reasoning about $deps$. On the other hand, within the method $Reset$, $U_{Clock}(p, \mathbf{this}, 0) = (time \leq 0)$ which does not follow from $p \in \mathbf{this}.deps$ and the precondition given for $Reset$ without information about $deps$.

$Reset$ should only be allowed if no clocks depend on this master, which would follow from $deps = \emptyset$ according to system invariant (5). We show our discipline for reasoning about $deps$ in the next subsection.

2.2 Notification of dependence

To maintain system invariant (5) we force each friend object to register itself with the granting object in order to include itself in the granting object's $deps$ field. Definition 2 of admissibility in Section 2.3 requires that Inv_T satisfy the following, for each pivot field g:

$$Inv_T(\mathbf{this}) \;\Rightarrow\; g = \mathbf{null} \;\vee\; \mathbf{this} \in g.deps \tag{9}$$

One way to satisfy (9) is to add $g = \mathbf{null} \;\vee\; \mathbf{this} \in g.deps$ as a conjunct of Inv_T.

We allow the field $deps$ to be updated only by the special statements $\mathbf{attach}$ and $\mathbf{detach}$ which add and remove an object from $\mathbf{this}.deps$.

$$\begin{array}{lcl} \mathbf{attach}\ E & \equiv & \mathbf{assert}\ E \neq \mathbf{null} \wedge \neg inv; \\ & & deps := deps \cup \{E\}; \end{array}$$

$$\begin{array}{lcl} \mathbf{detach}\ E & \equiv & \mathbf{assert}\ E \neq \mathbf{null} \wedge \neg E.inv \wedge \neg inv; \\ & & deps := deps - \{E\}; \end{array}$$

The attach and detach statements are allowed only in the code of the class T' where T' declares T to be a friend; their effect is to update $\mathbf{this}.deps$. It is in code of T' that we need to reason about $deps$ and thus to use $\mathbf{attach}$. This means that it is incumbent upon a friend to call some method in the granter when setting a pivot field to refer to the granter. This gives the granter a chance to either record the identity of the dependent (see the Subject/View example in Section 5.0) or to change some other data structure to reflect the fact that the dependent has registered itself (as in the Clock example, completed in Section 2.3).

Aside 2 *One could imagine that attach is triggered automatically by the assignment in a dependent to its pivot field. It is possible to work out such a system but it has the flaw that the granter is not given a chance to establish and maintain invariants about* $deps$.

Also, the conjunct $\neg E.inv$ in the precondition to **detach** *is stronger than necessary. The alternative is to require either that E is unpacked or that it no longer has its pivot field referring to* **this**, *but that would require the granter to know more about the pivot fields in its friends than we would like. In [NB04], we formulate the detach statement with the weaker pre-condition.*

2.3 Summary

To summarize the required annotations and program invariants, we begin with our original example from Figure 1 and rewrite it as shown in Figure 2. The two invariants in

```
class Master {
  time : int;
  invariant 0 ≤ time;
  friend Clock reads time;
  Master()
    ensures inv ∧ ¬comm;
  { time := 0; pack this; }
  Tick(n : int)
    requires inv ∧ ¬comm ∧ 0 ≤ n;
    modifies time;
    ensures time ≥ old(time);
  {
    unpack this;
    time := time + n;
    pack this;
  }
  Connect(c : Clock)
    requires inv;
    ensures c ∈ this.deps;
  {
    unpack this;
    attach c;
    pack this;
  }
}
```

```
class Clock {
  t : int;
  m : Master;
  invariant m ≠ null ∧ this ∈ m.deps;
  invariant 0 ≤ t ≤ m.time
  guard m.time := val by m.time ≤ val;
  Clock(mast : Master)
    requires mast ≠ null ∧ mast.inv;
    ensures inv ∧ ¬comm;
  {
    m := mast;
    t := 0;
    m.Connect(this);
    pack this;
    this.Sync();
  }
  Sync()
    requires inv ∧ ¬comm;
    modifies t;
    ensures t = m.time;
  {
    unpack this;
    t := m.time;
    pack this;
  }
}
```

Fig. 2. Clocks synchronized with a master clock. $Inv_{Clock}(\mathbf{this})$ depends on $\mathbf{this}.m.time$ but does not own $\mathbf{this}.m$.

the $Clock$ are conjoined to be the invariant for the class. In the constructor for $Clock$, t must be initialized to zero and the call to $m.Connect$ must occur in order to satisfy the class invariant before calling **pack**, but the call to $Sync$ is done after the call to **pack**

in order to fulfill the precondition of $Sync$. Note that Inv_{Clock} now satisfies (9) owing to the conjunct $\mathbf{this} \in m.deps$. This conjunct is established in the constructor by the invocation $m.Connect(\mathbf{this})$. In this case, $Connect$ is needed only for reasoning. In most friendship situations the granter needs some method for registering friends in order to maintain more information about them. An example is the $Master$ class revised to cater for $Reset$, in Section 4.0 and also all of the examples shown in Section 5.

To summarize our methodology, we first recall the rule for annotation of field update, (8). A separate guard U_f is declared for each field f on which a friend depends, so the rule is as follows.

$$\begin{array}{l}\mathbf{assert}\ \neg E.inv \ \wedge\ (\,\forall\, p \ \mid\ p \in E.deps \bullet\ \neg p.inv \ \vee\ U_f(p, E, D)\ );\\ E.f := D;\end{array}$$

It is straightforward to adapt this rule to cater for there being more than one friend class, or more than one pivot field of the same granter type but we omit the details (see [NB04]). For this paper, we disallow multiple pivots of the same type.

A friend may declare more than one update guard for a given f, in which case any one may be chosen for use at an update site. Each update guard

$$\mathbf{guard}\ g.f := \mathbf{val}\ \mathbf{by}\ U(\mathbf{this}, g, val);$$

gives rise to the proof obligation

$$inv\ \wedge\ U(\mathbf{this}, g, \mathbf{val})\ \Rightarrow\ (Inv_T(\mathbf{this}))^{g.f}_{\mathbf{val}}$$

All of the update guards for a particular field are guaranteed to maintain the friend's invariant. That means a granter can pick the most convenient update guard in order to discharge its proof obligation before it does a field update. The four auxiliary fields $inv, comm, owner, deps$ may all appear in method specifications and assertions, but they are updated only by special statements.

We refrain from repeating the definitions of $\mathbf{pack}$ and $\mathbf{unpack}$, which remain unchanged from Section 1. The $\mathbf{set\text{-}owner}$ statement needs to be revised: a friend may be granted access to $owner$, in which case there needs to be an update guard for $owner$ just like for ordinary fields:

$$\begin{array}{l}\mathbf{set\text{-}owner}\ E\ \mathbf{to}\ D\ \equiv\\ \quad\mathbf{assert}\ E \neq \mathbf{null} \wedge \neg E.inv \wedge (D = \mathbf{null} \vee \neg D.inv);\\ \quad\mathbf{assert}\ (\,\forall\, p \ \mid\ p \in E.deps \bullet\ \neg p.inv \ \vee\ U_{owner}(p, E, D)\ );\\ \quad E.owner := D;\end{array}$$

Note that if D is an object, it must be unpacked as its invariant is at risk when E becomes owned.

Definition 2 (admissible invariant). *An invariant $Inv_T(o)$ is* admissible *just if for every $X.f$ on which it depends, $f \not\equiv inv$, $f \not\equiv comm$, and either*

- *X is o (in the formula that means X is $\mathbf{this}$);*
- *X is transitively owned by o and $f \not\equiv deps$; or*

– X *is* $o.g$ *where field* g *(called a pivot) has some type* T' *that declares "friend* T *reads* f *".*

Moreover, the implication

$$Inv_T(\mathbf{this}) \;\Rightarrow\; g = \mathbf{null} \;\vee\; \mathbf{this} \in g.deps \tag{10}$$

must be valid.

There are easy syntactic checks for the ownership condition, e.g., it holds if X has the form $g.h.\ldots.j$ where each is a **rep** field, or if X is variable bound by $(\forall X \mid X.owner = o \bullet \ \ldots\ )$. Requirement (10) is met by including either the condition $\mathbf{this} \in g.deps$ or the condition $g = \mathbf{null} \vee \mathbf{this} \in g.deps$ as a conjunct of the declared invariant. (A fine point is that an admissible invariant should *only* depend on $deps$ in this way; see [NB04].) Although we do not use it in this paper, it is possible to have a **pivot** tag that marks the fields in the friend class that appear in the friend's invariant. Then there would be an easy syntactic process for imposing the requirement and allowing no other dependence on $deps$.

We extend the three *system invariants* (0–2) with a fourth invariant. Taken together, they ensure the following, for all o, T, f with $\mathbf{type}(o) = T$.

$$o.inv \;\Rightarrow\; Inv_T(o) \tag{11}$$

$$o.inv \;\Rightarrow\; (\forall p \mid p.owner = o \bullet\ p.comm\ ) \tag{12}$$

$$o.comm \;\Rightarrow\; o.inv \tag{13}$$

$$\begin{array}{c}\text{For every } T', g, p \text{ such that } \mathbf{type}(p) = T' \text{ and } Inv_{T'} \text{ depends on pivot } g \\ p.g = o \wedge p.inv \;\Rightarrow\; p \in o.deps\end{array} \tag{14}$$

It is really the first invariant that is the key to the entire methodology. It abstracts an object's invariant, preserving data encapsulation and allowing flexibility for reentrancy. The other invariants are all mechanisms needed in order to maintain (11) in the presence of inter-object dependence. The second and third work within ownership domains while our contribution adds cooperating objects.

3 Soundness

Consider any program annotated with invariants, friend declarations, and update guards satisfying the stated restrictions. We confine attention to the core methodology summarized in Section 2.3. Suppose that the obligations are met: the invariants are admissible and the update guard obligations are satisfied. Suppose also that every field update is preceded by the stipulated assertion, or one that implies it. We claim that (11–14) are system invariants, that is, true in every state. We refrain from formalizing precisely what that means, to avoid committment to a particular verification system or logic.

A detailed formal proof of soundness for the full methodology is given in a companion paper [NB04]. An informal argument has been given for the features already present in the previous Boogie papers [BDF+03a,LM03] and our methodology augments the preconditions used in those papers. We consider highlights for the new features.

Consider first the new invariant (14), and the statements which could falsify it.

- **pack** sets $p.inv$, but under the precondition $Inv(p)$, and by admissibility this implies $p \in o.deps$ for any o on which p has a friend dependence.
- **new** initializes $deps = \emptyset$ but also $inv = \mathbf{false}$. By freshness, no existing object has an owner or friend dependency on the new object.
- A field update $E.f := D$ can falsify it only if f is a pivot of E, but this is done under precondition $\neg E.inv$.
- **detach** removes an element from $\mathbf{this}.deps$ but under precondition $\neg\mathbf{this}.inv$.

Invariants (12) and (13) do not merit much attention as they do not involve the new fields and the new commands **attach** and **detach** do not involve inv or $comm$.

For (11), we must reconsider field update, $E.f := E'$, because $Inv_T(o)$ can have friend dependencies. By invariant (14), if o is a friend dependent on E, either $\neg o.inv$ or $o \in E.deps$. In the latter case, the precondition for update requires $U_f(o, E, E')$. The proof obligation for this update guard yields that $Inv_T(o)$ is not falsified by the update.

Both **attach** and **detach** have the potential to falsify (11) insofar as object invariants are allowed to depend on $deps$ fields. A local dependence on $\mathbf{this}.deps$ is no problem, owing to precondition $\neg\mathbf{this}.inv$. An admissible invariant is not allowed to depend on the $deps$ field of an owned object. What about friends? An admissible invariant is *required* to depend on $g.deps$ for each pivot g, but in a specific way that cannot be falsified by **attach** and that cannot be falsified by **detach** under its precondition. Finally, the **detach** E statement has the potential to falsify the consequent in (14), and this too is prevented by its precondition that either $\neg E.inv$ or E has no pivots referring to **this**. The intricacy of this interdependence is one motivation for carrying out a rigorous semantic proof of soundness.

4 Extensions

In this section, we present two extensions to our method. The first is a methodology for creating an invariant that eases the burden of reasoning about the $deps$ field in the granting class. The second is a syntactic extension to the update guard that provides extra information to the granting class after it performs a field update.

4.0 Tracking dependencies in invariants

We look again at the $Reset$ method in $Clock$. In order to set $time$ to zero, an instance of $Master$ must know either that each of the clocks referring to it have their value of t also as zero or that there are no clocks referring to it. Because of the proof obligation (9) on the class $Clock$, the latter case is true when $deps$ is empty. For this example, it suffices for the master clock to maintain a reference count, $clocks$, of the clocks that are referring to it via their field m, incrementing it each time **attach** is executed and decrementing it upon each **detach** statement. That is, variable $clocks$ maintains the invariant $clocks = size(deps)$. Given that invariant, the precondition for the update to $time$ in $Reset$ can be that $clocks$ is equal to zero.

In general, we refer to the invariant that the granting class maintains about its $deps$ variable as Dep. The invariant must be strong enough to derive enough information

about *all* objects $p \in deps$ to establish the precondition in (8). Thus we formulate Dep as a predicate on an element of $deps$ and introduce the following invariant as a proof obligation in the granting class.

$$(\forall\, p \mid p \in deps \bullet Dep(\mathbf{this}, p)) \tag{15}$$

As with U, we make $\mathbf{this}$ an explicit parameter in the declaration.

We extend the *friend* syntax in the granting class to define Dep:

$$\mathbf{friend}\ x : T\ \mathbf{reads}\ f\ \mathbf{keeping}\ Dep(\mathbf{this}, x)$$

It binds x in predicate Dep which may also depend on state visible in the granting class. The default is $Dep(\mathbf{this}, x) = \mathbf{true}$, easing the obligation but providing no help in reasoning about $deps$. Like any invariant, Dep cannot depend on inv or $comm$. In terms of the verification of the granting class, the effect is conjoin (15) to any declared invariant.

Figure 3 shows a version of $Master$ with $Reset$. Note that in the constructor, the value of $clocks$ must be set to zero in order to establish the "keeping" predicate, since initially $deps$ is empty. The preconditions for $Connect$ and $Disconnect$ restrict the value of $deps$ in order to keep an accurate count of the number of clocks referring to the master clock. Class $Clock$ need not be revised from Figure 2.

In this example, Dep is independent of the individual identities of the friend objects. The Subject/View example (Section 5.0) shows a more typical use of Dep.

4.1 Getting results from friendship

In contrast to the fixed pack/unpack/inv protocol which abstracts $Inv(p)$ to a boolean field, we have formulated the friend-invariant rule in terms of a shared state predicate. The associated methodology is to introduce public (or module-scoped) state variables with which to express U. (Minimizing the state space on which U depends could facilitate fast protocol checking as in Fugue [DF01,DF03].

Whereas invariants are invariant, states get changed. The proposal so far is that the public interface of the dependent class T should reveal information about changes relevant to T. Given that T publishes the condition under which shared state may be changed, why not also publish the effect of such changes?

We extend the update guard declaration to include predicate Y for the result state:

$$\mathbf{guard}\ g.f := \mathbf{val}\ \mathbf{by}\ U(\mathbf{this}, g, \mathbf{val})\ \mathbf{yielding}\ Y(\mathbf{this}, g, \mathbf{val});$$

The proof obligation on the friend class becomes

$$inv \ \wedge\ U(\mathbf{this}, g, \mathbf{val}) \ \Rightarrow\ (Inv_T(\mathbf{this}) \ \wedge\ Y(\mathbf{this}, g, \mathbf{val}))^{g.f}_{\mathbf{val}}$$

Note the resemblance to a pre/post specification in which the invariant is explicit.

At a field update site in the granting class, the yielding predicate can be depended on after the update:

$$\begin{array}{l}
\mathbf{assert}\ \neg E.inv; \\
\mathbf{assert}\ (\forall\, p \mid p \in E.deps \bullet \neg p.inv \vee U(p, E, D)\ ) \\
E.f := D \\
\mathbf{assume}\ (\forall\, p \mid p \in E.deps \bullet \neg p.inv \vee Y(p, E, D)\ )
\end{array}$$

```
class Master {
  time : int;
  clocks : int;
  invariant 0 ≤ time;
  friend c : Clock reads time keeping clocks = size(deps);
  Master()
    ensures inv ∧ ¬comm;
  { time := 0; clocks := 0; pack this; }
  Tick(n : int)
    requires inv ∧ ¬comm ∧ 0 ≤ n;
    modifies time;
    ensures time ≥ old(time);
  { unpack this; time := time + n; pack this; }
  Reset()
    requires inv ∧ clocks = 0;
    modifies time;
  { unpack this; time = 0; pack this; }
  Connect(c : Clock)
    requires inv ∧ c ∉ deps;
    modifies clocks;
    ensures c ∈ this.deps;
  { unpack this; clocks := clocks + 1; attach c; pack this; }
  Disconnect(c : Clock)
    requires inv ∧ c ∈ deps;
    modifies clocks;
    ensures c ∉ this.deps;
  { unpack this; clocks := clocks − 1; detach c; pack this; }
}
```

Fig. 3. Master clock with reset.

The predicates U and Y are likely to be useful in specifications of methods of T. Together with method specifications, the $\mathbf{guard}/\mathbf{yielding}$ statements of a class give the protocol by which it may be used.

5 Examples

In this section, we present several examples that demonstrate our methodology. We show some, but not all, details of their verification. The Subject/View example 5.0 demonstrates the use of our methodology for enforcing a behavioral protocol. In Section 5.1, the cooperation involves the use of a shared data structure. Finally, Section 5.2 illustrates how the *peer* concept[LM03] mentioned in Aside 1 can be easily encoded as a friendship relation.

5.0 Subject/View

In Figure 4, the class $Subject$ represents an object that maintains a collection of objects of type $View$ that depend on it. We refer to the object of type $Subject$ as the subject and each object that it holds a reference to in its collection vs as a view. In particular,

```
class Subject {
  val : int;
  version : int;
  rep vs : Collection⟨View⟩;
  friend v : View reads version, val keeping v ∈ this.vs

  void Update(n : int)
    requires inv ∧ ¬comm ∧ ( ∀ v ∈ vs • v.inv ∧ ¬v.comm ∧ Sync(v, this)); )
    modifies val, version;
    ensures val = n ∧ version = old(version) + 1 ∧ ( ∀ v ∈ vs • Sync(v, this) );
  {
    unpack this;
    version := version + 1;
    val := n;
    pack this;
    foreach v ∈ vs do v.notify();
  }
```

Fig. 4. The class $Subject$.

each view depends on the fact that whenever the state of the subject, represented by the val field (which could be a much more elaborate data structure), is changed in the method $Update$, then it will receive a call to its $Notify$ method. As part of its $Notify$ method, a view will make callbacks to its subject to retrieve whatever parts of

the updated state it is interested in. We do not show these state-querying methods (also known as *observers*).

To express the synchronization, the subject maintains a field $version$ which indicates the number of times that $Update$ has been called. A view also keeps track of a version number, vsn; a view is up to date if its version matches that of its subject.

In this example, the method $Update$ requires that the views be uncommitted so that they can be re-synchronized using their $Notify$ method. This is much easier to establish than the requirement that they be unpacked. For example, it is sufficient for the views to be peers of the subject, i.e., that they have the same owner.

Note that the subject packs itself before calling $Notify$ for all of its views. The views are then free to make state-observing calls on the subject, all of which presumably have a precondition that inv holds for the subject. Yet it is very important to realize that $Update$ is safe from re-entrant calls while it is in the middle of notifying all of the views, because a view would not be able to establish the pre-condition that all of the views are in sync with the subject. It is only *after* the method $Update$ has terminated that a view can be sure all of the views have been notified, and if it makes a re-entrant call, then that would come before $Update$ terminates.

The exception to this is if a view somehow knew that it was the only view for the subject. But in that case, a re-entrant call to Update does not cause any problems with the synchronization property. It still can lead to non-termination, but that is outside of the scope of our specification.

In Figure 5, the class $View$ publishes an update guard and update result for updates by the subject to its $version$ field, and an update guard without an update result for modifications to the subject's val field. The guards given are not the weakest possible, but rather are chosen to avoid exposing internal state. We define $Sync$ and Out as:

$$\begin{aligned} Sync(x : View,\ y : Subject) &\equiv x.vsn = y.version \\ Out(x : View,\ y : Subject) &\equiv x.vsn + 1 = y.version \end{aligned}$$

Even though the class $Subject$ uses $View$'s field vsn in the precondition and post-condition of $Update$, $View$ does not have to declare it as a friend class. However, the field must be accessible in the scope of the class $Subject$, e.g., by being public. To keep control of it, the class $View$ could define a read-only property [Gun00] and make the field itself private. We leave such details out of our examples. The invariant for the class is the conjunction of the two separately declared invariants.

The formal definitions for the update guards are:

$$\begin{aligned} U_{version}(x, y, z) &= Sync(x, y) \wedge z = x.vsn + 1 \\ U_{val}(x, y, z) &= x.vsn \neq y.version \end{aligned}$$

Note that because of the implication in Inv_{View}, the update guard for $s.val$ is written so as to falsify the antecedent; the guard is independent of z, which represents the value assigned to the field. This enforces a restriction on the order in which the $Subject$ can update the fields, even though the reverse order has equivalent effect. The $Subject$ must first update its $version$ field to make the implication vacuously true, and only then update its val field.

```
class View {
  private s : Subject;
  vsn : int;
  private cache : int;
  invariant s.version − 1 ≤ vsn ≤ s.version ∧ (vsn = s.version ⇒ cache = s.val);
  invariant s = null ∨ this ∈ s.deps;
  guard s.version := val by Sync(this, s) ∧ val = vsn + 1 yielding Out(this, s);
  guard s.val := val by vsn ≠ s.version;
  void Notify()
    requires ¬comm ∧ inv ∧ s.inv ∧ Out(this, s);
    ensures Sync(this, s);
    modifies vsn;
  {
    unpack this;
    vsn := vsn + 1;
    "read state from s" ; // This is why s.inv was required.
    pack this;
  }
}
```

Fig. 5. The class *View*.

Allowing the *View* to impose this requirement on *Subject* seems unfortunate, especially since the *Subject* has unpacked itself at the beginning of *Update* and so it would seem it should be able to update its fields in any order as long as it can re-establish its invariant before it tries to pack itself again. The example illustrates the price to be paid for the Boogie approach. Having the system invariants hold at "every semicolon" is conceptually simple and technically robust, but like any programming discipline this one disallows some programs that are arguably correct and well designed. If an inconsequential ordering of two assignments is the only annoyance then we are doing very well indeeed.

There are four proof obligations imposed by our methodology. In the granting class *Subject*, the assert before each of the two field updates in *Update* must be satisfied (8) and we have to show that the *Dep* predicate holds for every member of *deps* (15). That is, we have to show the following condition is invariant:

$$(\forall\, p \mid p \in deps \bullet p \in vs) \tag{16}$$

The obligations on the friend class *View* are that its advertised update guards maintain its invariant (7) and that it is in the *deps* field of the subject upon which it is dependent (9). We show the details of verifying the obligations only for the first three obligations.

To verify the assignment to $version$ in $Subject.Update$, the corresponding update guard from $View$ must be satisfied.

$$
\begin{array}{l}
(\,\forall\, p \;\mid\; p \in \mathbf{this}.deps \bullet\; \neg p.inv \;\vee\; U(p, \mathbf{this}, \mathbf{this}.version + 1)\,) \\
\equiv \{\text{Definition of guard}\} \\
(\,\forall\, p \;\mid\; p \in \mathbf{this}.deps \bullet\; \neg p.inv \;\vee\; (Sync(p, \mathbf{this}) \wedge \mathbf{this}.version + 1 = p.vsn + 1)\,) \\
\Leftarrow \{\text{Strengthening}\} \\
(\,\forall\, p \;\mid\; p \in deps \bullet\; Sync(p, \mathbf{this}) \wedge \mathbf{this}.version + 1 = p.vsn + 1\,) \\
\Leftarrow \{(16)\} \\
(\,\forall\, v \;\mid\; v \in vs \bullet\; Sync(v, \mathbf{this}) \wedge \mathbf{this}.version + 1 = v.vsn + 1\,) \\
\Leftarrow \{\, Update.\mathbf{requires} \;\Rightarrow Sync(p, \mathbf{this}) \wedge Sync(p, \mathbf{this}) \Rightarrow v.vsn = \mathbf{this}.version\} \\
(\,\forall\, v \;\mid\; v \in vs \bullet\; \mathbf{this}.version + 1 = \mathbf{this}.version + 1\,) \\
\Leftarrow \mathbf{true}
\end{array}
$$

This fulfills the proof obligation (8).

To verify the assignment to val in $Subject.Update$, we use the update guard in $View$ for val.

$$
\begin{array}{l}
(\,\forall\, p \;\mid\; p \in \mathbf{this}.deps \bullet\; \neg p.inv \;\vee\; U(p, \mathbf{this}, n)\,) \\
\equiv \{\text{Definition of guard}\} \\
(\,\forall\, p \;\mid\; p \in \mathbf{this}.deps \bullet\; \neg p.inv \;\vee\; p.vsn \neq \mathbf{this}.version\,) \\
\Leftarrow \{(16)\} \\
(\,\forall\, v \;\mid\; v \in vs \bullet\; \neg v.inv \;\vee\; v.vsn \neq \mathbf{this}.version\,) \\
\Leftarrow \{\text{pre-condition of } Update \text{ re: } Sync \text{ and update of } version\} \\
(\,\forall\, v \;\mid\; v \in vs \bullet\; \neg v.inv \;\vee\; v.vsn \neq v.vsn + 1\,) \\
\Leftarrow \mathbf{true}
\end{array}
$$

This fulfills the proof obligation (8).

In order to satisfy the proof obligation for (16), $Subject$ must provide a method in which **attach** is executed:

$$
\begin{array}{l}
\mathbf{void}\ Subject.Register(v : View) \\
\quad \mathbf{requires}\ \neg comm \wedge inv \wedge v \notin vs; \\
\quad \mathbf{ensures}\ v \in vs; \\
\quad \mathbf{modifies}\ vs; \\
\{ \\
\quad \mathbf{unpack\ this}; \\
\quad vs := vs + \{v\}; \\
\quad \mathbf{attach}\ v; \\
\quad \mathbf{pack\ this}; \\
\}
\end{array}
$$

Clearly, this makes (16) an invariant, since there are no other occurrences of **attach** that could modify the value of $deps$. For reasons of space, we do not show the code in $View$ that calls it.

Each update guard in $View$ must be shown to fulfill the obligation of (7), that its invariant will not be violated as long as the guard holds. Here we show only the update

guard for $version$, the one for val is even easier.

$$
\begin{array}{l}
(Inv_{View} \wedge Sync(\mathbf{this}, s) \wedge \mathbf{val} = \mathbf{this}.vsn + 1) \Rightarrow (Inv_{View})^{g.f}_{\mathbf{val}} \\
\equiv \{\text{Simplifying } Sync(\mathbf{this}, s) \wedge Inv_{View}\} \\
(\mathbf{this}.vsn = s.version \wedge \mathbf{val} = \mathbf{this}.vsn + 1) \Rightarrow (Inv_{View})^{s.version}_{\mathbf{val}} \\
\equiv \{\text{Substitution}\} \\
(vsn = s.version \wedge \mathbf{val} = vsn + 1) \Rightarrow \\
\qquad \mathbf{val} - 1 \leq vsn \leq \mathbf{val} \wedge (vsn = \mathbf{val} \Rightarrow cache = s.val) \\
\Leftarrow \{\text{Simplification}\} \\
\mathbf{true}
\end{array}
$$

5.1 Producer/Consumer

In this example, we show two objects that share a common buffer. There are two classes, $Producer$ and $Consumer$. Their definitions are shown in Figure 6 and Figure 7, respectively.

```
class Producer {
  buf : int[ ]; n : int; con : Consumer;
  invariant 0 ≤ n < buf.length;
  friend o : Consumer reads con, n, buf keeping o = con;

  Producer(b : int[ ])
    requires b ≠ null ∧ b.length > 1;
    ensures deps = ∅ ∧ inv ∧ ¬comm;
  { buf := b; n := 0; pack this; }
  void SetCon(c : Consumer)
    requires inv ∧ ¬comm ∧ c ≠ null ∧ deps = ∅;
    modifies con;
    ensures deps = {c} ∧ con = c;
  { unpack this; attach c; con := c; pack this; }
  void Produce(x : int)
    requires inv ∧ ¬comm ∧ |con.n − n| > 0;
    modifies n, buf;
  { unpack this; buf[n % buf.length] := x; n := (n + 1) % buf.length; pack this; }
}
```

Fig. 6. The class $Producer$.

We call instances of the former *producers* and instances of the latter *consumers*. A producer places elements into a circular buffer while consumers read them. Each object maintains a cursor into the common buffer; the producer can place more elements into the buffer as long as it does not overrun the consumer. Likewise, the consumer can only read elements from the buffer as long as its cursor does not overrun the producer's

```
class Consumer {
  buf : int[ ]; n : int; pro : Producer;
  invariant pro.con = this ∧ buf ≠ null;
  invariant pro.buf = buf ⇒ 0 ≤ |pro.n − n| < buf.length;

  guard pro.buf := val by false;
  guard pro.con := val by val = this;
  guard pro.n := val by 0 ≤ |n − val|;

  Consumer(p : Producer)
    requires p.inv ∧ ¬p.comm ∧ p.con = null;
    modifies p.con;
    ensures inv
  { buf := p.buf; pro := p; n := b.length − 1; pro.SetCon(this); pack this; }
  int Consume()
    requires inv ∧ ¬comm ∧ (n + 1) % buf.length < pro.n;
    modifies n;
  { unpack this; n := (n + 1) % buf.length; pack this; return(buf[n]); }
}
```

Fig. 7. The class *Consumer*.

cursor. The buffer is empty when the producer's cursor is one element ahead (modulo the buffer length) of the consumer's cursor. When the both cursors are equal, then the buffer is full. Because of this encoding, the buffer's length must be greater than one and one slot in the array is never used. (The particular slot is not constant, but the capacity of the buffer is one less than the number of slots.)

It is important to note that this is not a full specification of the functional behavior of the two classes. The specification is only of the synchronization between the two, just as was done for the Subject/View example. For schematic patterns this is especially useful; the specification can be combined with a particular usage of the pattern to fill out the details.

The class *Consumer* is given friend access to *buf*, *con*, and *n*. Being given access to *buf* does not give the consumer the right to depend on the contents of *buf* in its invariant. Such a dependence would be a dependence path of length two: one step to the *buf* and the next to the sub-object at some index *i*. We do not allow this; we allow only direct dependence on a pivot field.

Someone familiar with reasoning about arrays is tempted to view the assignment to an element of buf in *Produce* as an assignment to the entire array with a new array that is the same at all points except for the updated element. If that were the case, then the update in *Produce* would have to satisfy the update guard for *buf* in *Consumer*. However, the update is to a sub-object of *buf*; the only difference is that the field "name" is a number rather than an identifier. It is only updates to the actual field itself that must satisfy the update guard.

The friend access for buf is given to the consumer because it needs to make sure the producer does not update the field to a new, different buffer. This is expressed by the update guard for $pro.buf$ being **false**. It is possible to allow the producer to change its buffer, either by requiring that the buffer is empty, or even to allow the consumer to continue reading from the old buffer as long as the producer no longer is using it. However, we do not consider these more advanced scenarios.

The update guard for con is slightly different: it allows the producer to modify the field, but only to assign the consumer to it. The update guard for the producer's cursor, n, allows the producer to fill as many slots as are available, even though in this particular implementation, the producer fills only one slot at a time.

We do not show the proofs for the field updates in $Producer$; all of the proofs are immediate.

5.2 Doubly-linked list with transfer

For our last example, we consider a doubly-linked list. The class $List$ with its $Insert$ and $Push$ methods is shown in Figure 8. Each $List$ object has a reference to an object

```
class List {
  head : Node;
  invariant head = null ∨ (head.prev = null ∧ head.owner = this);
  void Insert(x : int)
    requires x > 0 ∧ inv ∧ ¬comm;
    modifies ;
    ensures ;
  {
    n : Node; n := new Node(x); this.Push(n);
  }
  void Push(n : Node)
    requires inv ∧ ¬comm;
    requires n ≠ null ∧ n.prev = null ∧ n.next = null;
    requires n.inv ∧ ¬n.comm ∧ n.owner = null;
    modifies head, n.comm, n.owner;
    ensures n.comm ∧ n.owner = this;
  {
    unpack this;
    set-owner n to this;
    if (head = null) head := n; else head := head.Insert(n);
    pack this;
  }
}
```

Fig. 8. The class $List$. The design caters for a method to be added in Figure 12.

of type $Node$; the nodes have forward and backward references to other $Node$ objects.

In [LM03], this example serves to explain the concept of *peers*: objects who share a common owner. Remember by Definition 2 that an object's invariant is allowed to depend on its peers. The class $Node$ is shown in Figure 9. Because each of the nodes that

```
class Node {
  val : int;
  prev : Node;
  next : Node; // pivot field
  friend n : Node reads prev, owner keeping n = prev
  invariant 0 < val ∧ prev ≠ this ∧
    (next = null ∨ (next.owner = owner ∧ next.prev = this));
  Node(x : int)
    requires 0 < x;
    ensures val = x ∧ inv ∧ prev = null ∧ next = null;
  {
    val := x;
    prev := null;
    next := null;
    pack this;
  }
}
```

Fig. 9. Part of the class $Node$. Other methods are in subsequent Figures.

are linked into a particular list share the same owner, if a node is able to pack and unpack itself, then it is also able to do that for any other node in the list. In terms of our methodology, this means no update guards are needed. Instead, the recursive friend access is needed so that a node's invariant can depend on the node to which its $next$ field points. The **keeping** clause maintains that a node keeps a reference to its friend in its $prev$ field. Thus the quantification in the precondition for field update can be simplified by the one-point rule. Notice that within the outer "else" clause of $Insert$ (Figure 10), we unpack the argument n so that we can assign to its pivot field $next$ without worrying about violating $Inv_{Node}(n)$. All of the conditions required before packing it back up are met through a combination of the (rather elaborate) pre-conditions on the method and the assignments that take place in the body of the method. We do not show the details; all of the required conditions are immediately present.

To add realistic complications to the code, the list is maintained in ascending order and if desired this could be expressed using node invariants, again avoiding reachability expressions.

Figure 12 shows an example of transferring ownership. In this case, the first node in one list is moved over to another list, s. It is important to see that it transfers the actual object of type $Node$, as well as the contents of the node. The helper function, $Disconnect$ (Figure 11), removes a node from the entanglements of the pointers in its list and maintains $deps$.

```
class Node {
  Node Insert(n : Node)
    requires inv ∧ ¬comm;
    requires n ≠ null ∧ n.val > 0 ∧ n.next = null ∧ n.prev = null;
    requires n.inv ∧ ¬n.comm;
    requires prev = null ∨ prev.val ≤ n.val;
    requires owner = n.owner;
    modifies next*.next, next+.prev;
    ensures result ≠ null ∧ result.prev = old(this.prev);
  {
    result : Node;
    unpack this;
    if (n.val ≥ val) {// insert after self
      if (next = null) {// this is the last node
        next := n;
      } else {// pass it down the line
        next := next.Insert(n);
      }
      unpack next;
      next.Attach(this);
      pack next;
      result := this;
    } else {// insert before self
      unpack n;
      n.next := this;
      this.Attach(n);
      pack n;
      result := n;
    }
    pack this;
     return result;
  }
}
```

Fig. 10. The method *Insert* in class *Node*.

```
class Node {
  void Disconnect()
    requires inv ∧ prev = null ∧ ¬comm ∧ next ≠ null ∧ next.inv;
    ensures next = null ∧ old(next).prev = null;
    modifies next, next.prev, next.deps;
  {
    unpack this;
    unpack next;
    next.Detach(this);
    pack next;
    next := null;
    pack this;
  }
}
```

Fig. 11. The *Disconnect* method in class *Node*.

6 Related Work

The most closely related work is that of Leino and Müller [LM03] which uses an explicit *owner* field that holds a pair (o, T) of the owner together with the type T at which o has a relevant invariant. The paper by Müller et al. [MPHL03] lucidly explains both the challenges of modular reasoning about object invariants and the solution using ownership. They prove soundness for a system using types to formulate ownership, based on Müller's dissertation [Mül02] which deals with significant design patterns in a realistic object-oriented language. They also discuss the problem of dependence on non-owned objects and describe how the problem can be addressed soundly by ensuring that an object's invariant is visible where it may be violated; thus sound proof obligations can be imposed, as is developed further in [LM03]. Section 1 has reviewed [LM03] and the other Boogie paper [BDF+03a] at length and we encourage the reader to consult them for further comparisons.

The Extended Static Checker for project, especially ESC/Modula-3 [DLNS98,LN02,FLL+02], treats object invariants by what Müller [Mül02] calls the visibility approach which requires invariants to be visible, and thus liable for checking, wherever they may be violated. This can significantly increase the number of proof obligations for a given verification unit and the focus of the work is on mitigation by abstraction. An idiom is used for expressing invariants as implications $valid \Rightarrow \ldots$ where *valid* is an ordinary boolean field, serving like *inv*.

Liskov, Wing, and Guttag [LG86,LW94] treat object invariants but in a way that is not sound for invariants that depend on more than one object. There has been a lot of work on alias control to circumscribe dependency. Ownership type systems [CNP01,Cla01] explicitly address the problem of encapsulating representation objects on which an invariant may sensibly depend. Much of this line of work struggles to reconcile efficient static checking with the challenges of practical design patterns. Boyapati, Liskov and Shrira [BLS03] argue that their variation on ownership types achieves encapsulation

```
class List {
  void TransferHeadTo(s : List)
    requires s ≠ this ∧ head ≠ null ∧ ¬comm ∧ inv ∧ ¬s.comm ∧ s.inv;
    modifies s.?;
    ensures ;
  {
    unpack this;
    n : Node; n := head; head := head.next;
    if (n.next ≠ null) n.Disconnect();
    set-owner n to null;
    pack this;
    s.Push(n);
  }
}
```

Fig. 12. The method $TransferHeadTo$ in class $List$.

sufficient for sound modular reasoning but they do not formalize reasoning. They exploit the semantics of inner objects in Java which provides a form of $owner$ field but suffers from semantic intricacies and precludes ownership transfer.

Banerjee and Naumann [BN02a] use a semantic formulation of ownership in terms of heap separation and show that it ensures preservation of object invariants. They focus on two-state invariants, i.e., simulation relations, to obtain a representation independence result (for this purpose, read access by clients is restricted). The ownership property is enforced by a static analysis that does not impose the annotation burden of ownership types but like ownership types it requires the ownership invariant to hold in every state. A version has been developed that includes transfer of ownership, but it depends on a static analysis for uniqueness and the proof of soundness was difficult [BN03]. The representation-independence theorem states that the invariant of a class T is preserved by clients *if* it is they are preserved by methods of T. The theorem allows invocation of state-mutating methods on pointers outgoing from encapsulated representation objects, including reentrancy. Unlike work such as [MPHL03], the problem of verifying methods of T is not addressed.

Separation logic [Rey02] can express very directly that a predicate depends only on some subset of the objects in the heap. It has successfully treated modular reasoning about an object invariant in the case of a single class with a single instance [OYR04]. Some of the attractive features are achieved in part by a restriction to a low-level language without object-oriented features. This is an exciting and active line of research and it will be interesting to see how it scales to specifications and programs like those in Section 5.

```
class Node {
  void Attach(n : Node)
    requires ¬inv ∧ n ≠ null;
    ensures prev = n ∧ n ∈ deps;
    modifies deps, prev;
  { attach n; prev := n; }
  void Detach(n : Node)
    requires ¬inv ∧ n ≠ null ∧ ¬n.inv;
    ensures prev = null ∧ n ∉ deps;
    modifies deps, prev;
  { detach n; prev := null; }
}
```

Fig. 13. The *Attach* and *Detach* methods in class *Node*.

7 Conclusions

Formal systems for programming must always cope with the conflict between the flexibility real programs display and the restrictions formal analysis demands. Our work extends Boogie's system for object invariants to cope with a real-world situation: dependence across ownership boundaries. We have constructed a protocol that imposes minimal obligations upon the participating classes; it is inevitable that there are some extra verification conditions. In addition, we have tried to maintain Boogie's mantra: hiding private implementation details while providing explicit knowledge about the state of an object's invariant. Our contribution is a workable system for specifying and verifying cooperating classes.

While one approach would be to allow, or even to insist, for cooperating classes to be knowledgeable about each other's private implementation state, we believe that is important to provide for as much abstraction as possible. The protocols could all be expressed in terms of more abstract properties instead of concrete fields allowing a class implementation to change without disturbing its friend classes.

Our presentation has left out all mention of sub-classing, but the actual definitions have all been made taking it into account.

There are many ways in which we plan to extend our work. For instance, our methodology could be presented independently from ownership. Currently, we think it best to use ownership where possible and thus it is important that friendship fits well with ownership. We also need to explore the use of static analysis for alias control in common cases.

Our update guards are related to *constraints* [LW94]; it would be interesting to formulate them as constraints, thus shifting more of the burden to the granting class instead of the friend class.

We will continue to explore different design decisions to weaken the obligations. The tradeoffs are between being able to easily verify the specifications and code against allowing the most flexibility for the programmer.

We are implementing our scheme as part of the Boogie project. Empirical evaluation will doubtless point out many problems and opportunities for improvement.

Acknowledgements

We would like to thank Rustan Leino (not to mention K. Rustan and M. Leino) for all of his comments and help. Wolfram Schulte made many helpful suggestions, especially pointing out the connection between update guards and constraints. Anindya Banerjee and Rob DeLine made helpful expository suggestions.

References

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

[BDF+03a] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Formal Techniques for Java-like Programs 2003*, July 2003. Available as Technical Report 408, Department of Computer Science, ETH Zurich. A newer version of this paper is [BDF+03b].

[BDF+03b] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. Manuscript KRML 122b, December 2003. Available from http://research.microsoft.com/~leino/papers.html.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[BN02a] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. Extended version of [BN02b]. Available from http://www.cs.stevens-tech.edu/~naumann/oceri.ps, 2002.

[BN02b] Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *POPL*, pages 166–177, 2002.

[BN03] Anindya Banerjee and David A. Naumann. Ownership transfer and abstraction. Technical Report TR 2004-1, Computing and Information Sciences, Kansas State Univ., 2003. http://www.cs.stevens-tech.edu/~naumann/otranseTR2004-1.pdf.

[BS03] Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, 2003.

[CD02] David Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, November 2002.

[CL02] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

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

[CNP01] David G. Clarke, James Noble, and John M. Potter. Simple ownership types for object containment. In Jørgen Lindskov Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, 2001.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

[DF03] Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software baroque? Available from http://research.microsoft.com/~maf/papers.html, 2003.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University, 2001.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[Gun00] Eric Gunnerson. *A Programmer's Introduction to C#*. Apress, Berkeley, CA, 2000.

[Jon83] Cliff B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

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

[LM03] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. Manuscript KRML 132, December 2003. Available from http://research.microsoft.com/~leino/papers.html.

[LN02] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.

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

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

[MPHL03] P. Müller, A. Poetzsch-Heffter, and G.T. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zürich, Chair of Software Engineering, October 2003.

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

[NB04] David A. Naumann and Mike Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). Submitted; available from http://www.cs.stevens-tech.edu/~naumann/tim.pdf, 2004.

[OYR04] P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.

[Rey02] John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.