Using History Invariants to Verify Observers

K. Rustan M. Leino and Wolfram Schulte

Microsoft Research, Redmond, WA, USA
{leino,schulte}@microsoft.com

Abstract. This paper contributes a technique that expands the set of object invariants that one can reason about in modular verification. The technique uses *history invariants*, two-state invariants that describe the evolution of data values. The technique enables a flexible new way to specify and verify variations of the observer pattern, including iterators. The paper details history invariants and the new kind of object invariants, and proves a soundness theorem.

1 Introduction

The *observer* pattern is an important and common programming idiom [13]. For example, it is a foundation of the model-view-controller paradigm on which all modern graphical user interfaces rely. The observer pattern consists of a *subject* object, which contains some data that may change over time, and a number of *observer* objects. An observer *depends on* the data of the subject in some way. For example, an observer may display the current data values of the subject in a graphical user interface. For efficiency, such an observer may keep a local copy of the data to be displayed, so that it can redraw the display without needing to consult the subject. A variation of the observer pattern is the *iterator* pattern [13], where the subject is a collection and the observers are iterators. An observer may iterate through the items of the collection, providing clients with one data item at a time. These two patterns are different mainly in that the collection does not have references to its iterators. In this paper, we focus on the one-to-many dependency between the subject and observers, which the two patterns have in common, so we will simply refer to both of them as the observer pattern.

To verify the correctness of a program that uses the observer pattern, it is necessary to be able to write specifications for both subject and observers. We are interested in *modular verification* of programs, which allows a program's modules (or classes) to be verified separately. In order for the verification process to be *sound*, the separately verified correctness of each module should imply the correctness of the whole program. For the observer pattern, this means we want to be able to specify and verify the subject separately from the observers.

Verifying the observer pattern is a challenge. The difficulty is that the data consistency of an observer, which is expressed as an *object invariant*, depends on the data of the subject. Updates of the subject and the maintenance of these invariants must therefore be coordinated. The situation is further complicated by the fact that the subject may not be able to reach (through object references in the heap) all the observers, and the observer invariants, let alone the observer classes, may not be available in the separate-verification context of the subject. A partial solution, which works when the observers are known by the subject, has been given by Barnett and Naumann [5].

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 80–94, 2007.

In this paper, we introduce a specification and verification methodology that is well-suited for supporting the kinds of object invariants one wants to write in observer classes. In a nutshell, the subject advertises how its data values evolve over time, and this allows observers to declare object invariants that depend on the subject's data, provided the object invariants are insensitive to the evolution of the subject. In more detail, our solution consists of the following ingredients:

1. We use *history invariants* to specify how an object may evolve. A history invariant is a reflexive and transitive *two-state predicate* that relates any earlier state to any later state in a program's execution. In our solution, subjects have history invariants.
2. We allow an object invariant of an observer to access the fields of the subject, provided the dereference goes via a field annotated with a new field modifier, **subject**. If an object invariant dereferences a **subject** field, we call it an *observer invariant*.
3. We explicitly keep track of whether an object invariant is known to hold, in which case we say that the object is *consistent*.
4. An observer invariant can be assumed if the observer and its subject are *both* in the consistent state.
5. For the soundness of modular verification, each observer invariant gives rise to an additional proof obligation, which is that it be maintained under the history invariant of the subject.

Our main contributions in this paper are 2, 4, and 5, which together give a methodology to specify and verify observer patterns, including its iterator variation. Ingredient 3 comes from the Boogie methodology, which we explain in Section 2. For ingredient 1, history invariants were introduced by Liskov and Wing [22] under the name of *constraints*, and are supported by the Java Modeling Language (JML) [18]; our paper contributes a formalization of history invariants in the presence of reentrancy and representation objects.

Example. Figure 1 shows our solution to specifying a verifiable observer pattern. An observer's cache depends on the state of the subject. When a subject's state is updated, it notifies all of its observers, so that they can synchronize their caches.

We use a field $vers$ (for "version") in both the subject and observers, so that an observer can detect whether it is currently synchronized with the subject. We have found this specification idiom useful for all of our observer-pattern examples, though our methodology does not depend on it. (The $vers$ field is in fact used in the implementation of the iterator pattern in both .NET [1] and Java [14], where it is used to detect modifications of the underlying collection when there is still an active iterator.)

Note that between the update of $state$ and $vers$ in method $Update$, the observer's invariant is broken. Our methodology handles this on account of ingredient 4. At the end of the **expose** block, the observer's invariant holds again, on account of the specification idiom used in the observer invariant.

The program is correct and satisfies the proof obligations of our methodology: the history invariants are admissible, because they are reflexive and transitive; the updates performed by the $Subject$ methods are allowed, because they maintain the history invariants; and the observer invariants are admissible, because they are maintained under the subject's history invariants.

```
interface IObserver {
  void Notify();
}

class Subject {
  rep Set⟨peer IObserver⟩ obs;
  int state;  int vers;

  history invariant old(vers) ⩽ vers;
  history invariant vers = old(vers) ⇒
      state = old(state);

  Subject()
  { initialize (this) {
      state = 0;  vers = 0;
      obs = new Set⟨peer IObserver⟩();
    }
  }

  void Register(IObserver o)
    requires o ≠ null ∧ o.owner = owner;
  { expose (this)
      { obs.Add(o); }
    o.Notify();
  }

  void Update(int y)
  { expose (this)
      { state = y;  vers = vers + 1; }
    foreach (IObserver o in obs)
      { o.Notify(); }
  }

  int Get()
    ensures result = state;
  { return state; }
}
```

```
class MyObserver : IObserver {
  readonly subject Subject subj;
  int cache;  int vers;

  invariant vers ⩽ subj.vers;
  invariant
    subj ≠ null ∧ subj.vers = vers ⇒
      cache = subj.state;

  MyObserver(Subject s)
    requires s ≠ null;
    ensures owner = s.owner;
  { initialize (this) {
      cache = s.Get();  vers = s.vers;
      sub = s;  owner = s.owner;
    }
  }

  void Notify()
  { expose (this) {
      cache = s.Get();
      vers = s.vers;
    }
  }

  void DisplayData()
  { ... }
}


class Program {
  void Main() {
    Subject s = new Subject();
    MyObserver o =
        new MyObserver(s);
    s.Register(o);
    s.Update(57);
  }
}
```

Fig. 1. An example of the observer pattern, where class *Subject* uses objects of type *IObserver* as its observers. Each of the two columns in this figure is a separately verifiable module. The details of the constructs used in this example are explained in the paper. As details that make the verification go through, we have assumed that each object has a reference valued *owner* and a boolean *inv* field. Further, we assumed that the condition $PeerConsistent(x) \wedge \neg x.owner.inv$ is implicitly added as a postcondition to all constructors (with **this** for x), as a precondition to all methods (with **this** for x), and as a precondition to all constructors and methods (for each reference parameter x). On entry to a constructor body, we also assume that the new object starts off with some arbitrary, unshared, and exposed owner. Finally, we assume that all methods are implicitly allowed to modify the fields of **this** and of any parameter x, and also the fields of the peers of **this** and x.

Outline. In the next section, we describe the foundations of our work, as well as a body of previous work that tackles the problem of specifying and verifying the observer pattern. In Section 3, we define history invariants and their associated proof obligations. In Section 4, we define the additional machinery needed to support observer invariants, culminating in a soundness theorem about them. The paper wraps up with additional examples (Section 5), more related work (Section 6), future work (Section 7), and conclusions (Section 8).

2 Methodologies for Object Invariants

In this section, we review how a modular-verification system deals with objects invariants. We also look at how previous work has tackled the problem of specifying and verifying the observer pattern. In this section and throughout most of the paper, we ignore the issue of subclassing.

Visible-state semantics. The first question to address when designing a methodology for object invariants is: when does the invariant of an object hold? A simple answer is: whenever no constructor or method of the object is active. This simple methodology is called *visible-state semantics* [25,18], because an object's invariant holds in all states visible to public clients of the object.

Because of the possibility of reentrancy in object-oriented programs, we need to be concerned about the situation where an object a breaks its invariant, calls a method on an object b, and then b calls back into some method of a that assumes the invariant to hold. Visible-state semantics prevents this situation by using alias control, as with the *universe type system* [25,26]: a can be used only as a read-only object while the method on b is invoked, restricting b's use of a to read-only methods, and visible-state semantics does not allow read-only methods to rely on the invariant.

Boogie methodology. A richer methodology is the *Boogie methodology* supported by Spec# [4]. The basic Boogie methodology [2] adds a bit inv to every object. If $inv = true$, the object is said to be *consistent*, its invariant holds, and its fields are not allowed to be updated. If $inv = false$, the object is said to be *mutable*, its invariant may be violated, and the fields are allowed to be updated. This guarantees the following *program invariant* (a condition that holds in all reachable states of the program):

$$(\forall o \bullet\ o.inv \Rightarrow Inv(o)) \tag{1}$$

where, here and throughout, the quantification ranges over non-null, allocated objects and $Inv(o)$ denotes the declared object invariant of o. For the moment, we assume $Inv(o)$ to be an *intra-object invariant*, that is, that it depends only on the fields declared in the class of o.

By mentioning inv explicitly in preconditions, methods can indicate whether or not they expect the object invariant to hold on entry.

The Boogie methodology controls changes to the inv field by introducing two special program statements. The statement **unpack** o changes $o.inv$ from $true$ to $false$, and the statement **pack** o changes $o.inv$ from $false$ to $true$, after first checking that $Inv(o)$ holds. (This check can be done either by static verification or by run-time checking. In this paper, we focus on static verification.)

Use of **unpack** and **pack** is typically stylized, so in this paper we instead use a block statement **initialize** (o) $\{S\}$, which abbreviates:

$$S; \ \ \mathbf{pack}\ o$$

and a block statement **expose** (o) $\{S\}$, which abbreviates:

$$\mathbf{unpack}\ o; \ \ S; \ \ \mathbf{pack}\ o$$

The former typically wraps the body of a constructor and the latter wraps the bodies of other methods, as we have seen in Fig. 1.

Owners and representation objects. Going beyond intra-object invariants, we now consider invariants that span several objects. To meet preconditions involving inv, it becomes necessary for an object o to know the state of its *representation objects* (or *rep objects*), that is, the objects that o uses in its implementation. The Boogie methodology lets a class declare a field with the **rep** modifier to say that the field references a rep object (*cf.* [8,6,7,25,10]).

We introduce another field for every object, $owner$, which determines an ownership hierarchy among objects [19]. The $owner$ field points in the inverse direction of **rep** fields; in fact, declaring a field f to be **rep** induces the object invariant:

$$\mathbf{this}.f = \mathbf{null} \lor \mathbf{this}.f.owner = \mathbf{this}$$

The methodology guarantees the following program invariant [2,19]:

$$(\forall\, o \bullet\ o.inv \Rightarrow (\forall\, r \bullet\ r.owner = o \Rightarrow r.inv\,)) \tag{2}$$

To achieve this guarantee, the methodology restricts assignments to $owner$. For our purposes, it suffices to set $owner$ upon creation of objects (see [19] for a treatment of ownership transfer) and to add the following precondition to the **unpack** o statement: $\neg o.owner.inv$.

Using ownership, we can allow object invariants to dereference **rep** fields. That is, if f is a **rep** field, then we can now allow $Inv(o)$ to depend on $o.f.x$ for any field x. Nevertheless, this is not sufficient for the observer pattern: an observer can mention fields of its subject (like $\mathbf{this}.subj.x$) in its object invariant only if $subj$ is a **rep** field, which implies the observer is the unique owner of the subject. Not only does this disallow the existence of more than one observer, but it also seems odd for an observer to consider its subject to be part of its implementation.

Peers. As another possible field modifier, the Boogie methodology allows **peer** [25,19,10]. Declaring a reference-valued field f to be **peer** induces the following object invariant:

$$\mathbf{this}.f = \mathbf{null} \lor \mathbf{this}.owner = \mathbf{this}.f.owner$$

Unlike **rep** fields, **peer** fields are not allowed to be freely dereferenced in object invariants. However, **peer** modifiers lead us to the useful concept of an object o being *peer consistent*, which says that o and all its peers are consistent:

$$PeerConsistent(o) \ = \ (\forall\, p \bullet\ p.owner = o.owner \Rightarrow p.inv\,)$$

A subject and its observers are better suited as peers rather than that one owns the other, because if both use $\mathit{PeerConsistent}(\mathbf{this})$ in their method preconditions, then the subject methods can invoke methods on any observer, and vice versa.

Visibility-based invariants. To specify and verify the observer pattern, we need a methodology that allows us to mention $\mathbf{this}.subj.x$ in the invariant of observers, where $subj$ is a field that references the subject object and x is a field of the subject. This is allowed under the two restrictions of *scope visibility* [19].

The first restriction of scope visibility says that an observer can mention $\mathbf{this}.subj.x$ in its invariant if the invariant is visible to every verification context that can contain an update of the x field. This works out fine for the iterator pattern, but forbids the development of observer classes separate from the development of the subject class.

The second restriction is that updating a subject's field $s.x$ requires not only that the subject s be in the mutable state ($inv = \mathit{false}$), but also that every observer o for which $o.subj = s$ be in the mutable state. This restriction is hard to live with if the number of such observers o is unbounded. It is especially hard to live with if the observers are not reachable from the subject, which is the case in the iterator pattern.

Update guards. Barnett and Naumann relax the second restriction for visibility-based invariants [5]. Instead of requiring observers whose invariants mention $\mathbf{this}.subj.x$ to be in the mutable state when x is updated, Barnett and Naumann propose checking that the imminent update of x maintains the actual invariant of these observers. To provide some way to abstract over an observer's invariant, they also introduce the declaration of an *update guard* in the observer classes. The update guard is a condition on the update of the subject's x field that is sufficient to maintain the observer's invariant. The update guard is declared as a two-state predicate. For example, an update guard

$$\mathbf{this}.subj.x : \quad \mathbf{old}(\mathbf{this}.subj.x) \leqslant \mathbf{this}.subj.x$$

says that increasing the subject's x field maintains the observer's invariant.

Update guards can be used to specify the observer pattern, as long as the first restriction for visibility-based invariants holds: observer classes must be visible to the subject when it is verified.

Monotonicity. Another situation where we can allow an object invariant to mention $\mathbf{this}.f.x$ is when x is a read-only field. This situation is almost like for intra-object invariants, because if x is immutable, then the only way to change the value of $\mathbf{this}.f.x$ is to change $\mathbf{this}.f$. Immutability is a special case of monotonicity. If the value of a field x only changes monotonically, by some metric, then it is unproblematic to allow an invariant $\mathit{Inv}(o)$ to mention $o.f.x$, provided $\mathit{Inv}(o)$ is maintained under such monotonic changes (*cf.* [11]). Monotonicity conditions can be specified as reflexive and transitive history invariants, which is in fact what we do.

Our solution. Let us briefly compare our solution to the previous work we have discussed in this section. Rather than declaring update guards in the observer classes, which requires these observer classes to be known when the subject's data are updated, we propose declaring *in the subject class* how the subject's data may evolve. This means that the subject need not be aware of how many observers and observer classes there are—such an observer is allowed to declare an invariant that *depends on* the subject's data, provided the invariant has the property that it is automatically maintained when the subject's data evolve as advertised.

3 History Invariants

History invariants (or *constraints*, as Liskov and Wing called them [22]) are two-state predicates. In this section, we first discuss intra-object history invariants in the context of a visible-state semantics, and then look into inter-object history invariants in the context of the Boogie methodology.

Visible-state semantics. In the visible-state semantics, an object invariant for object o is a property that should hold of all visible states of o. A history invariant for o is a property that should hold for any earlier-later pair of visible states of o. History invariants can therefore be used to constrain the way that values change over time.

The history invariant in the following example says that the value of $size$ will only ever increase:

```
class Histogram⟨K⟩ {
  int size;
  invariant 0 ≤ size;
  history invariant old(size) ≤ size;
  ...

  Histogram(int size)
    requires 0 ≤ size;  {...}

  void Resize(int size)
    requires this.size ≤ size;
  ...
```

Let's see how the $Histogram$ class maintains its history invariant. The object's first visible state is defined at the time the $Histogram$ constructor finishes. Different, subsequent visible states can be created only by mutating methods, like $Resize$. The pre- and post-states of $Resize$ are visible states. Consequently, a visible-state semantics for $Histogram$ has to guarantee that the history invariant for **this** also holds between pre- and post-states of $Resize$.

For visible-state semantics, history invariants are thus added as proof obligations to post-conditions of public methods. But note that their verification only guarantees that each pair of method pre- and post-states obeys the history invariant. However, history invariants for an object o have to hold between any two visible states that result from a computation on o. By requiring history invariants to be reflexive and transitive, we guarantee that the history invariant holds between any earlier and later visible states.

Boogie methodology. We now describe how to incorporate history invariants into the Boogie methodology. Continuing our example, we could implement the $Histogram$ class using a **rep** field of type $Hashtable$, where we assume that the class $Hashtable$ has a $size$ field:

```
class Histogram⟨K⟩ {
  rep Hashtable⟨K, int⟩ ht;
  invariant 0 ≤ ht.size;
  history invariant
    old(ht.size) ≤ ht.size;
  ...

  Histogram(int size)
    requires 0 ≤ size;  {...}

  Resize(int size)
    requires ht.size ≤ size;
  ...
```

In the visible-state semantics above, a history invariant of an object holds for pairs of its visible states. In the Boogie methodology, a history invariant of an object holds for pairs of its consistent states.

In the following formulas, we adorn state-dependent predicates with stores as indices. One-state predicates have one state, two-state predicates have two states as indices, *i.e.*, $q_{\sigma,\tau}$ denotes q evaluated in the two states σ, τ where **old** expressions in q refer to state σ and the non-old expressions refer to state τ. We use $\mathit{Hist}(o)$ to denote the declared history invariant of o; $[\mathit{Hist}(o)]_{\sigma,\tau}$ is $\mathit{Hist}(o)$ evaluated in the two states σ, τ. We use $\sigma \leqslant \tau$ to denote that state σ occurs earlier than state τ in a program run.

For the rest of the paper, we only allow ownership-based invariants with **rep** fields. These give rise to the program invariants (1) and (2). The methodology extended with history invariants also needs to establish the following program invariant:

$$(\forall\, o, \sigma, \tau \bullet\ \sigma \leqslant \tau \wedge [o.inv]_\sigma \wedge [o.inv]_\tau \Rightarrow [\mathit{Hist}(o)]_{\sigma,\tau}\ ) \tag{3}$$

This important condition says that if σ and τ are two states that occur in that execution order and $o.inv$ holds in both of those states, then the history invariant for o relates those two states.

We define a history invariant to be *admissible* if (a) it is reflexive, (b) it is transitive, and (c) it depends only on the fields of **this** and the fields of transitive rep objects of **this**. While property (c) is just a syntactical check, properties (a) and (b) give rise to the proof obligations:

$$(\forall\, o, \sigma \bullet\ [\mathit{Hist}(o)]_{\sigma,\sigma}\ ) \tag{4}$$

$$(\forall\, o, \sigma, \tau, \upsilon \bullet\ [\mathit{Hist}(o)]_{\sigma,\tau} \wedge [\mathit{Hist}(o)]_{\tau,\upsilon} \Rightarrow [\mathit{Hist}(o)]_{\sigma,\upsilon}\ ) \tag{5}$$

which are checked by a theorem prover.

In addition to the proof obligations stemming from admissibility, a history invariant also needs to be verified at various points in the program. Since the Boogie methodology enforces that a field $t.f$ can be changed only if t and all its transitive owners are mutable, the only way to violate the condition (3) in a program is when an object o changes (in τ) from mutable to consistent and there was a previous time (namely σ) when o was consistent. Therefore, we check history invariants at the end of **expose** blocks. That is, we redefine **expose** $(o)\ \{S\}$ to stand for:

$$\text{let } \rho = \ \sigma \text{ in }\ \mathbf{unpack}\ o;\ \ S;\ \ \mathbf{assert}\ [\mathit{Hist}(o)]_{\rho,\sigma};\ \ \mathbf{pack}\ o$$

where we use σ to denote the current program state.

We can now prove that our methodology for history invariants is *sound*, that is, that (3) follows from the admissibility checks and the added check in the **expose** statement.

Proof (3). Consider the (possibly infinite) sequence of states in any execution of the program, and consider a particular object o. Consider any two states σ and τ in this sequence, such that $o.inv$ holds in both of those states. The proof now proceeds by induction over the length of the sequence from σ to τ. We consider four cases.

- If σ and τ are the same state, then $[\mathit{Hist}(o)]_{\sigma,\tau}$ follows directly from reflexivity (4).
- If σ and τ are different states and there is some intervening state ρ in which $o.inv$ also holds, then by the induction hypothesis on the two shorter sequences, $[\mathit{Hist}(o)]_{\sigma,\rho}$ and $[\mathit{Hist}(o)]_{\rho,\tau}$ hold, so $[\mathit{Hist}(o)]_{\sigma,\tau}$ holds by transitivity (5).

- If σ and τ are consecutive states, then σ and τ bracket some primitive statement. We argue that this primitive statement does not affect any field $x.f$, where x is o or a transitive rep object of o, because the methodology allows a field update of $x.f$ only if x and its transitive owners are mutable (see (1) and (2)).
- If σ and τ are different, non-consecutive states and they have no intervening state in which $o.inv$ holds, then σ and τ bracket the execution of an $\mathbf{expose}\ (o)$ statement. The added check in the $\mathbf{expose}$ statement guarantees that $[Hist(o)]_{\sigma,\tau}$ holds. □

4 Observer Invariants

Object invariants of observers often depend on the stability of subjects. A prime example for this dependency is given by the observer pattern, as implemented in Figure 1. Its observer invariant says: if the version of the observer coincides with the version of the collection, then the cache of the state of the observer coincides with the state held in the subject. This property can now be used, for example, by the observer's $DisplayData$ method: without reading the subject's entire state, it can now guarantee that it displays the current value of the subject, provided the versions of subject and observer still agree.

Observers make the dependency on their subject explicit by annotating a field with the $\mathbf{subject}$ modifier. Declaring a field $subj$ to be $\mathbf{subject}$ induces the object invariant:

$$\mathbf{this}.subj = \mathbf{null} \lor \mathbf{this}.subj.owner = \mathbf{this}.owner$$

This is the same as the object invariant induced by $\mathbf{peer}$ fields, but $\mathbf{subject}$ fields will be used differently in defining the admissibility condition for object invariants.

We define an object invariant to be *admissible* if (a) it depends only on fields of $\mathbf{this}$, fields of transitive rep objects of $\mathbf{this}$ (that is, fields like $\mathbf{this}.f_0.f_1.\cdots.x$ where the f_i are $\mathbf{rep}$ fields), and fields of subject objects of $\mathbf{this}$ (that is, fields like $\mathbf{this}.subj.x$, where $subj$ is a $\mathbf{subject}$ field), and (b) it is stable under the history invariant of any subject object dereferenced in the invariant. While property (a) is just a syntactic check, property (b) gives rise to the following proof obligation, for every $\mathbf{subject}$ field $subj$ that is dereferenced in the invariant:

$$\begin{array}{l}(\forall\, o, \sigma, \tau \bullet \\ \quad \sigma \leqslant \tau \land [o.inv]_\sigma \land (\forall f \bullet\ [o.f]_\sigma = [o.f]_\tau\ ) \land \\ \quad [o.subj.inv]_\sigma \land [o.subj.inv]_\tau \land [Hist(o.subj)]_{\sigma,\tau} \\ \quad\ \Rightarrow [Inv(o)]_\tau\ )\end{array} \qquad (6)$$

This condition is checked by the theorem prover.

In the presence of $\mathbf{subject}$ fields, the object invariant doesn't necessarily hold when the object is consistent (as we saw at the program point between the updates of $state$ and $vers$ in method $Update$ in Fig. 1). However, it does hold if the object's subject objects are consistent as well. So, in our methodology, the program invariant (1) is replaced by the following program invariant:

$$\begin{array}{l}(\forall\, o \bullet\ o.inv \land \\ \ (\forall \mathbf{subject}\ \text{field}\ f\ \text{of}\ o\ \text{dereferenced in}\ Inv(o) \bullet\ o.f = \mathbf{null} \lor o.f.inv\ ) \\ \qquad\quad \Rightarrow Inv(o)\ )\end{array} \qquad (7)$$

(To receive the benefit of a stronger program invariant, one can think of $Inv(o)$ as denoting just one conjunct of the object invariant, which reduces the number of f's that the antecedent says need to be consistent, and then repeat the program invariant for each conjunct of the object invariant.)

In order for (7) to hold, we need to add an additional check as part of the $\mathbf{pack}$ statement, namely: for every subject field f of o, $\mathbf{pack}\ (o)$ also imposes the precondition $o.f = \mathbf{null} \vee o.f.inv$.

We can now prove that our revised methodology is *sound*, that is, that (7) follows from the admissibility checks and the added preconditions of the $\mathbf{pack}$ statement. For brevity, we will give the proof for an object invariant $Inv(o)$ that mentions exactly one $\mathbf{subject}$ field, $subj$.

Proof (7). The proof runs by induction over the sequence of states in any execution of the program. The induction base is trivial: Program execution starts in a state where no objects are allocated. In the induction step, we consider the different ways in which a state change could violate (7):

case o is allocated: A newly allocated object o start with $\neg o.inv$.
case a heap location $t.x$ that is referred to by a term $o.f_0.f_1.\cdots.x$ in $Inv(o)$ is changed: According to the methodology, a field $t.x$ is allowed to be updated only if t and its transitive owners are mutable, so $\neg o.inv$.
case $o.inv$ is changed from *false* to *true* (which happens in $\mathbf{pack}\ (o)$): The precondition of the pack statement checks that $Inv(o)$ holds.
case $o.inv$ holds and $s.inv$ is changed from *false* to *true* (which happens in $\mathbf{pack}\ (s)$), for an s such that $o.subj = s$: We distinguish two cases:

– If this $\mathbf{pack}\ (s)$ was part of an $\mathbf{initialize}\ (s)$, then $\neg s.inv$ always held before this time. But since $o.inv$ holds, there must have been an earlier $\mathbf{pack}\ (o)$, $o.subj$ would have been unchanged since the most recent such $\mathbf{pack}\ (o)$, and that $\mathbf{pack}\ (o)$ would have checked that $o.subj.inv$ held. So this case does not exist.
– If this $\mathbf{pack}\ (s)$ was part of an $\mathbf{expose}\ (s)$, then let σ denote the state immediately before the $\mathbf{expose}\ (s)$ and let τ denote the state immediately after $s.inv$ has been set to true, *i.e.*, after the $\mathbf{pack}\ (s)$. Due to the block structure of expose statements, we know that the condition $\neg s.inv$ is stable throughout the execution after state σ and before state τ. Moreover, $o.inv$ is stable between these states, because any change to $o.inv$ would mean there was a $\mathbf{pack}\ (o)$ inside the $\mathbf{expose}\ (s)$, and that $\mathbf{pack}\ (o)$ would have checked $s.inv$, which doesn't hold. Because $o.inv$ is stable, then so is $o.f$ for every field f of o. In summary, we now have:

$$\begin{array}{l}\sigma \leqslant \tau \wedge [o.inv]_\sigma \wedge (\forall f \bullet\ [o.f]_\sigma = [o.f]_\tau\ ) \wedge \\ {[o.subj.inv]_\sigma} \wedge [o.subj.inv]_\tau\end{array}$$

By the last two conjuncts and (3), we also have $[Hist(o.subj)]_{\sigma,\tau}$. Altogether, we then have the antecedent of (6), from which we conclude $[Inv(o)]_\tau$. □

5 Further Examples

We show two more examples of how to use history invariants to prove observer patterns.

Collection Iterator Pattern [13]. Figure 2 shows an application of our methodology to the class of a *Collection* (the subject) and its associated class of *Iterator* objects

```
class Collection⟨T⟩ {
  rep T[] elems;
  int ct;  int vers;

  invariant elems ≠ null ∧
    0 ⩽ ct ⩽ elems.Length;
  history invariant
    old(vers) ⩽ vers;
  history invariant
    vers = old(vers) ⇒
      ct = old(ct) ∧
      elems[0 : ct] = old(elems[0 : ct]);

  Collection(int capacity)
    requires 0 ⩽ capacity;
  { initialize (this) {
      elems = new T[capacity];
      ct = 0;  vers = 0;
    }
  }

  void Add(T t)
  { expose (this) {
      if (ct = elems.Length) { ... }
      elems[ct] = t;
      ct++;  vers++;
    }
  }

  T Remove(int i)
    requires 0 ⩽ i < ct;
  { T t = elems[i];
    expose (this) {
      elems[i: ct − 1] = elems[i + 1: ct];
      ct−−;  vers++;
    }
    return t;
  }
}
```

```
class Iterator⟨T⟩ {
  readonly subject Collection⟨T⟩ coll;
  readonly int vers;
  int n;  bool inRange;

  invariant coll ≠ null ∧
    −1 ⩽ n ∧ vers ⩽ coll.vers;
  invariant
    vers = coll.vers ⇒
      inRange = (0 ⩽ n < coll.ct);

  Iterator(Collection⟨T⟩ c)
    requires c ≠ null;
    ensures owner = c.owner;
  { initialize (this) {
      coll = c;  vers = c.vers;
      n = −1;  inRange = false;
      owner = c.owner;
    }
  }

  bool MoveNext()
    requires vers = coll.vers
      otherwise InvalidOperation;
    ensures result = inRange;
  { expose (this) {
      if (n < coll.ct) { n++; }
      inRange = n < coll.ct;
    }
    return inRange;
  }

  T Current()
    requires vers = coll.vers
      otherwise InvalidOperation;
    requires inRange;
  { return coll.elems[n]; }
}
```

Fig. 2. Class *Collection*⟨*T*⟩ represents a list of items of type *T* that can be retrieved by an *Iterator*⟨*T*⟩. These classes exhibit a variation of the observer pattern and their specifications are handled by our methodology.

(the observers). Each *Collection* object contains a *vers* field that is increased with each update of the collection. The iterator's methods require as a precondition that the versions of the iterator and collection match up. If they don't match up, the caller is in error, a situation that is caught when trying to statically verify the caller.

```
class Master {
  int tm;  int vers;

  invariant 0 ⩽ tm;
  history invariant old(vers) ⩽ vers;
  history invariant vers = old(vers) ⇒
      old(tm) ⩽ tm;

  Master()
    ensures tm = 0 ∧ vers = 0;
  { initialize (this)
      { tm = 0;  vers = 0; }
  }

  void Tick(int n)
    requires 0 ⩽ n;
    ensures old(tm) ⩽ tm;
  { expose (this)
      { tm = tm + n; }
  }

  void Reset()
    ensures tm = 0;
  { expose (this)
      { vers = vers + 1;  tm = 0; }
  }
}
```

```
class Clock {
  readonly subject Master ms;
  int tm;  int vers;

  invariant ms ≠ null ∧ 0 ⩽ tm;
  invariant vers ⩽ ms.vers;
  invariant vers = ms.vers ⇒
      tm ⩽ ms.tm;

  Clock(Master m)
    requires m ≠ null;
    ensures owner = m.owner;
  { initialize (this) {
      ms = m;  Synch();
      owner = m.owner;
    }
  }

  private void Synch()
  { tm = ms.tm;  vers = ms.vers; }

  int GetTime()
    ensures 0 ⩽ result ⩽ ms.tm;
  { if (vers ≠ ms.vers)
      { expose (this) { Synch(); } }
    return tm;
  }
}
```

Fig. 3. Our rendition of Barnett and Naumann's master and slave clock example [5]. For verification, we assume the private method *Synch* to be inlined at its call sites.

For compatibility with existing non-verified clients, the iterator methods will throw an *InvalidOperation* exception in case the *Iterator* client is in error.

Note that the observer invariant is necessary for verifying the definedness of the method *Current*: The implicit precondition says that the iterator is peer consistent. The collection is a peer of the iterator, since *coll* is declared with **subject**, so peer consistency of the iterator implies peer consistency of the collection. Because the iterator and collection are both consistent, the observer invariant can be assumed on entry to *Current*. Together with the explicit preconditions of the method, we conclude that the array index n in *Current*'s implementation is in range.

Master and Slave Clocks [5]. A master clock has two timer functions, *Tick*, which increases the time, and *Reset*, which resets the time to zero. A slave clock's time never exceeds its master's time. Slaves have a *GetTime* method that returns the time at which the slave clock most recently synchronized its time with the master. The number of necessary synchronizations of a slave clock with a master clock should be minimal. This means that as long as *Tick* is called on the master, a slave doesn't have to synchronize.

But as soon as the master's clock is reset, a slave's clock must be synchronized to fulfill its contract. Figure 3 shows our solution.

6 Related Work

Automated program verification has a long history, *cf.* [23]. Only much more recently did it become feasible to do large-scale automatic reasoning as automatic theorem provers made great progress and are now optimized for proving software checking (*e.g.*, [9]), verification-condition generation became optimized for those theorem provers (*e.g.*, [12]), and programming methodology progressed (*e.g.*, [2,19,5,15]).

History invariants were introduced by Liskov and Wing [22] to constrain the behavior of possible subtypes. Their paper did not explore the possibility of using them for verifying object invariants. History invariants are also supported by the Java Modeling Language (JML) [18], which uses visible-state semantics. To the best of our knowledge, static verification tools for JML do not yet support history invariants.

Our use of history invariants is similar to Rely/Guarantee style reasoning as introduced by Jones [16]. It enables a compositional reasoning about concurrent programs. Rely/Guarantee conditions are also two-state predicates. In our setting, Rely/Guarantee conditions would mean that a subject guarantees the stability of a property on which the invariants of the observers rely.

Verifying observers is a form of verifying heap properties. This area has recently gotten a lot of attention (*e.g.*, [21]). In the sequel, we focus only on traditional program verification work for modern languages.

Another approach to specifying the update-notify idiom of the observer pattern is proposed by Middelkoop *et al.* [24]. They use a mix between the visible-state semantics and the Boogie methodology where all objects are consistent on method boundaries unless explicitly stated otherwise. The approach does not yet address representation objects.

Inspector methods [15] are pure methods that can depend on owned state. They elegantly address the existing data abstraction problem in ownership systems, but do not help in verifying observers independent from subjects.

Kassios's dynamic frames [17] abstractly specify the effect of mutator methods using abstraction functions and dependency relations (and without needing a built-in ownership system). The work is formulated in the context of an idealized logical framework; it was not developed to address maintaining observer invariants, but rather to delineate change. We look forward to seeing an implementation of the approach in an automatic program verifier.

Like observers and subjects, the classes of a program can depend on each other in a one-to-many way. For example, many classes depend on the $String$ class. A different approach exists for handling this situation [20].

An important recent strand in verifying heap structures is separation logic [27]. It is an extension of Hoare logic for programs that use pointers or references into a heap. However, its assertion language is not first order; instead, it uses a powerful spatial conjunction that is integral for partitioning the heap. While proof system for separation logic have been started, they are still somewhat primitive and tool support is not yet there for a full object-oriented language.

7 Future Work

We are currently investigating the best way to incorporate history invariants into Spec# [4] and the Boogie program verifier [3]. We want to further develop the presented methodology to support subtyping, which we believe to be an orthogonal issue, just like in the basic Boogie methodology [2]. With subtyping, one might have a situation where a subclass acts like an observer to a field declared in a superclass. Another area of interest is to understand how the verification of history invariants fits in with other methodologies, like monotonic type states [11] and visibility-based invariants. Last but not least, we want to explore whether history invariants can be used to verify more design patterns, like invariants over static fields.

8 Conclusion

This paper extends the limits of sound modular verification for inter-object invariants. In most previous approaches for one-to-many dependencies, all classes had to be developed together. Our approach allows one object (the subject) to export a history invariant, which other objects (the observers) can depend on. A history invariant typically describes some stability of the subject's state space. Introducing those properties has two benefits: it allows observers to make their validity dependent on the stability of the subject, and subjects do not have to know anything about the existence of observers. This fosters modular development and verification.

Acknowledgments. We are grateful to the anonymous referees for their thoughtful and helpful suggestions.

References

1. Brad Abrams. *.NET Framework Standard Library Annotated Reference, Volume 1.* Addison Wesley Longman Publishing, 2004.
2. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *JOT*, volume 3, number 6, pages 27–56, 2004.
3. Mike Barnett, Robert DeLine, Bart Jacobs, Bor-Yuh Evan Chang, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, September 2006.
4. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
5. Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, LNCS, pages 54–84. Springer, July 2004.
6. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.
7. Dave G. Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 292–310. ACM, November 2002.

8. Dave G. Clarke, John. M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, volume 33, number 10 in *SIGPLAN Notices*, pages 48–64. ACM, October 1998.
9. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
10. Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *FOOL/WOOD '07*. ACM SIGPLAN, January 2007. 13 pages.
11. Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *Proceedings of International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, July 2003.
12. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL 2001*, pages 193–205. ACM, January 2001.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.
14. James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.
15. Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In *FTfJP 2006*, July 2006.
16. Cliff B. Jones. Development methods for computer programs including a notion of interference. Technical report, Oxford University, PhD thesis, 1981.
17. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006*, volume 4085 of *LNCS*, pages 268–283. Springer, August 2006.
18. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
19. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 491–516. Springer, June 2004.
20. K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *LNCS*, pages 26–42. Springer, 2005.
21. Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS 2000*, pages 280–301, 2000.
22. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
23. D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.
24. Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Invariants for non-hierarchical object structures. In *Brazilian Symposium on Formal Methods, SBMF 2006*, pages 233–248. SBC, September 2006.
25. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002. PhD thesis, FernUniversität Hagen.
26. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 2006. To appear.
27. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL 2005*, pages 247–258. ACM, January 2005.