Class-Local Object Invariants

K. Rustan M. Leino
Microsoft Research
Redmond, WA, USA
leino@microsoft.com

Angela Wallenburg
Chalmers University of Technology
Göteborg, Sweden
angelaw@cs.chalmers.se

ABSTRACT

The correctness of object-oriented programs relies on object invariants. A system for verifying such programs requires a systematic method for coping with object invariants that can be violated temporarily. This paper describes a sound methodology for flexibly changing data locally in object structures, supporting programming patterns that occur frequently in practice. In more detail, to handle subclasses, previous approaches have been geared toward programs that update the fields of an object only in overridable virtual methods of the object. The enhanced methodology in this paper handles field updates in a much more flexible way. The flexibility can be applied to a field in the common case where the field is not mentioned in subclass invariants.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Class invariants; Programming by contract; Assertion checkers*

General Terms

Languages, Verification

Keywords

Specification, Verification, Object-oriented programming, Subclassing, Tool support, Automation

0. INTRODUCTION

Computer programs would not work if their variables always could take on arbitrary values. They work because programmers design their implementation to confine the values to meet certain constraints. For example, a binary-tree implementation might be designed to keep the data values in a sorted infix order. In object-oriented programming, the data consistency is described by *object invariants*. Object invariants do not always hold in a program; there are intermediate states during operations when the invariants are temporarily violated. When verifying programs, it is important to be precise about when the object invariants are expected to hold. In this paper, we describe a methodology for specifying and verifying object-oriented programs that takes advantage of the fact that many object invariants are violated only within a small scope.

ISEC'08, February 19-22, 2008, Hyderabad, India.


```
class Car {
  int speed;
  invariant 0 ⩽ speed;
  ...
}
class LuxuryCar extends Car {
  int cruiseControlSetting;
  invariant cruiseControlSetting = −1
      ∨ speed = cruiseControlSetting;
  ...
}
```

Figure 0: Fragments of two example classes. Class Car uses the `invariant` keyword to specify an object invariant that constrains its $speed$ field to be non-negative. Car's subclass $LuxuryCar$ declares an additional field, $cruiseControlSetting$, and declares an invariant that specifies a relation between it and $speed$.

Consider the example in Fig. 0 where class Car introduces a field $speed$ and declares with an object invariant that $speed$ is to be non-negative. A program verifier must check that updates of $speed$ maintain the invariant.

We are interested in *modular verification*, where one verifies a class without having the program text of either the clients or the subclasses. Modular verification enables the verification of program libraries and makes verification scalable.

To illustrate the interplay between modular verification, object invariants, and subclassing, consider the example in Fig. 0. Subclass $LuxuryCar$ declares an invariant that mentions the inherited superclass field $speed$, thus further constraining $speed$ for $LuxuryCar$ objects. If a program updates the $speed$ field of a $LuxuryCar$ object, then the verifier must check that the (Car invariant and) $LuxuryCar$ invariant is maintained. Because of polymorphic subtyping, the static type of the $LuxuryCar$ object being updated might be just Car. Without the program text for $LuxuryCar$, a modular verification of Car must instead rely on that the unknown parts of the program follow some systematic rules.

The *Boogie methodology* [1, 13, 3, 14, 9, 10, 16, 11, 15] provides such systematic rules for modular verification. To achieve modular verification, the methodology needs to restrict how fields are updated (we give details in Sections 1 and 2). Because the Boogie methodology allows invariants to mention fields declared in super-

classes, its restrictions on field updates in effect come down to this: if a field f is declared in a non-sealed class (that is, in a class that admits further subclasses), then updates of f must be performed by a virtual method.

In our experience with Spec# [2], which implements the Boogie methodology, we have found that a large number of field updates do not occur in virtual methods. The Spec# static program verifier (called Boogie [0]) reports errors for such updates, thus revealing places where the code has not been designed to cater for subclass extensions. Verifying the programs requires introducing virtual methods or declaring classes to be sealed. However, in many cases, these remedies may be heavy-handed or inappropriate, because the fields being updated are not intended to be further constrained by subclass invariants. An extreme, but common, example of this occurs with private fields, which cannot be mentioned outside the declaring class. In short, the fact that the Boogie methodology allows invariants to mention superclass fields comes at a high price.

We propose a more liberal set of rules for updating fields. These rules apply when invariants do not mention fields declared in superclasses. More precisely, the rules are more liberal for updates of fields that are known not to be mentioned in any subclass invariant, for instance if the fields are declared to be private. For fields that may be mentioned in subclass invariants, we retain the classic rules of the Boogie methodology (with one exception, which regards dereferencing fields of fields, see Section 6).

Our methodology applies to modern object-oriented languages, including Java and C#. We have implemented the methodology in Spec# and its static program verifier, Boogie. We have found that this makes it much easier to write verifiable programs and to verify existing programs. By making the new rules the default, programs can pay the price of the classic Boogie methodology only for those fields whose use is intended to be refined by subclasses.

We begin in Section 1 by introducing the details of our liberal rule set. In Section 2, we incorporate the classic rules in a new guise. Then, in Section 3, we give some examples that show what is gained by our enhanced methodology. We explain our implementation encoding in Section 4 and reflect on our experience with the enhanced methodology in Section 5. We show how to incorporate the classic rules for *ownership*, which allows aggregate objects whose invariants can span several objects (Section 6), and wrap up the paper with related work (Section 7) and conclusions (Section 8).

1. LOCAL FIELDS AND INVARIANTS

We consider a Java-like object-oriented language with single-inheritance classes. The class hierarchy is rooted at a built-in class called **object**. We think of the fields of an object as being partitioned into *class frames*, with one class frame for every superclass of its allocated type. For example, an object of allocated type $LuxuryCar$ in Fig. 0 has three class frames: **object**, Car, and $LuxuryCar$.

In this section, we restrict our attention to object invariants that only mention fields declared in the enclosing class. We call object invariants that adhere to that restriction *admissible*. The Car invariant in Fig. 0 is admissible, because $speed$ is declared in the class Car, but the $LuxuryCar$ invariant is not admissible, because $speed$ is declared in $LuxuryCar$'s superclass.

To keep track of whether or not object invariants hold, we follow the Boogie methodology and introduce two states for every class frame of an object, *mutable* and *valid*. If a class frame T of an object o is mutable, then the fields of o declared in class T are allowed to be updated and the invariant declared in T may be violated. If the class frame is valid, then the invariant holds and the fields are not allowed to be updated.

```
class Car {
  int speed;
  int windResistance;
  invariant 0 ⩽ speed
    ∧ windResistance = K * speed * speed;

  void SetSpeed(int kmph) {
    expose (this at Car) {
      speed = kmph;
      windResistance = K * speed * speed;
    }
  }
}
```

Figure 1: An example class with an invariant that constrains the value of $windResistance$ to be proportional to the square of $speed$ (where K is some constant whose definition we omit). The expose statement marks a block of code that updates the fields and eventually re-establishes the invariant.

We encode the two states by adding a boolean field $valid$ to every class frame. For an object o and a class frame T, we write $(o, T).valid$ to refer to this field. For convenience, we write $(o, T).mutable$ as a synonym for $\neg(o, T).valid$. When all class frames of an object are valid, we say that the object is *consistent*.

Next, we introduce rules that in every program state maintain the condition

$$(\forall\, o, T \bullet\ (o, T).valid\ \Rightarrow\ Inv_T(o)\ ) \qquad (0)$$

where o ranges over non-null, allocated objects and T ranges over class names, and where we write $Inv_T(o)$ to stand for the invariant of o declared in class T. We call a condition that holds in every program state a *program invariant*.

To change the value of $valid$, we follow the Boogie methodology and use an **expose** statement:

$$\textbf{expose}\ (o\ \textbf{at}\ T)\ \{S\}$$

changes $(o, T).valid$ from **true** to **false**, then executes statement S, and finally resets $(o, T).valid$ to **true**. Before resetting $valid$, the **expose** statement checks that $Inv_T(o)$ holds. Typically, T is the static type of o, but in this paper we show it explicitly.

The example in Fig. 1 shows the Car class with a method that updates $speed$. The method uses an **expose** statement to obtain the permission to modify the two fields. Note that the code might violate the invariant in the intermediate state between the two assignments. The **expose** statement checks that the object invariant holds again at the closing curly brace.

Formal Encoding.

To describe the various state transitions in detail, we represent the "open curly brace" and "close curly brace" of the statement **expose** $(o$ **at** $T)$ $\{S\}$ with the operations **unpack** o **from** T and **pack** o **as** T, respectively. The meaning of the unpack and pack operations are as follows:

```
unpack o from T ≡
  assert o ≠ null ∧ (o, T).valid;
  (o, T).valid := false
```

$$
\begin{array}{l}
\textbf{pack}\ o\ \textbf{as}\ T\ \equiv \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge (o, T).mutable; \\
\quad \textbf{assert}\ Inv_T(o); \\
\quad (o, T).valid := \textbf{true}
\end{array}
$$

where we use **assert** statements to denote conditions that are to be checked. It is an unrecoverable error if an asserted condition does not hold. We use the symbol $:=$ to distinguish the assignments above from ones contained in the source program.

When an object is allocated, all of its class frames start off in the mutable state. The end of a constructor in a class T implicitly packs the object being constructed for T:

$$\textbf{pack this as}\ T$$

where **this** denotes the object being constructed.

Finally, to enforce the rule that only fields of mutable class frames can be updated, we add a precondition to field updates:

$$
\begin{array}{l}
o.f = E\ \equiv \\
\quad \textbf{assert}\ o \neq \textbf{null} \wedge (o, T).mutable; \\
\quad o.f := E
\end{array}
$$

where f is a field declared in a class T.

It is instructive at this time to give a proof sketch that these rules do achieve program invariant (0). We look at four state changing operations: allocation, unpack, pack, and field update.

- The object-allocation operation increases the range of o in (0). Since a newly allocated object starts off with all class frames in the mutable state, program invariant (0) is maintained.

- The unpack operation changes $(o, T).valid$ to **false**, which falsifies the antecedent of (0). Hence, program invariant (0) is maintained.

- The pack operation changes $(o, T).valid$ to **true**, but does so only if $Inv_T(o)$ holds (that is, the pack operation asserts that $Inv_T(o)$ holds). Hence, program invariant (0) is maintained.

- From our definition of admissible invariant, the only object invariant that can be violated by a field update $o.f = E$ is $Inv_T(o)$ where T is the class that declares f. But since the update proceeds only if $(o, T).mutable$, program invariant (0) is maintained.

Note that the definition of admissible invariant is crucial to the argument above. Under the rules we presented in this section, condition (0) would not hold if, say, an object invariant could mention fields declared in superclasses or fields of fields. For example, since updates of $c.speed$ only require $(c, Car).mutable$, not $(c, LuxuryCar).mutable$, program invariant (0) would not be guaranteed if the $LuxuryCar$ invariant in Fig. 0 would be admissible. In Sections 2 and 6, we accommodate such invariants, respectively, by amending the definition of admissible invariant; the amended definitions will come at the price of more complicated rules.

Writing Method Specifications.

Program invariant (0) lets us conclude that the T invariant for an object o holds from knowing that the T frame of o is valid. Let us see what that means for program verification and method specifications.

Consider the $TicketManager$ class in Fig. 2. The following

```
class TicketManager {
  int[] tickets;
  int n;
  invariant tickets ≠ null ∧ 0 ⩽ n ⩽ length(tickets);

  TicketManager()
    ensures (this, TicketManager).valid;
  {
    tickets = new int[0];
    n = 0;
  }

  void AddTicket(int t)
    requires (this, TicketManager).valid;
  { ... }

  int DispenseTicket()
    requires (this, TicketManager).valid;
  {
    int t = −1;
    if (n < length(tickets)) {
      t = tickets[n];
      expose (this at TicketManager) { n = n + 1; }
    }
    return t;
  }
}
```

Figure 2: An example class showing a ticket manager that keeps a queue of service tickets, represented by integers. The first n tickets of the queue have been serviced. Tickets can be added to the queue using the $AddTicket$ method, whose implementation we have omitted. The $DispenseTicket$ method returns an unserviced ticket from the queue, if any, or -1 otherwise. The correctness of the array dereference $tickets[n]$ relies on the specified object invariant. The requires and ensures keywords introduce pre- and postconditions, respectively, and the fixed length of an array is retrieved by the function $length$.

code fragment shows how this class can be used:

```
TicketManager tm = new TicketManager();
tm.AddTicket(32);
tm.AddTicket(27);
int t = tm.DispenseTicket();
```

Note that this code does not need to know the $TicketManager$ invariant in detail: to prove the preconditions of $AddTicket$ and $DispenseTicket$, it suffices to know that the constructor returns with tm in the valid state (and that the other method calls do not change $valid$, a condition we ignore in this paper).

Let us also consider the verification of the implementation of $TicketManager$.

Since newly allocated objects start off with all class frames mutable, we can assume $(\textbf{this}, TicketManager).mutable$ on entry to the constructor, which meets the preconditions of the two field updates. An implicit **pack this as** $TicketManager$ operation ends the constructor. Since **this** is mutable at that time and the $TicketManager$ invariant has been established, the precondition of the pack operation is met and $(\textbf{this}, TicketManager).valid$ is set to **true**, thus establishing the declared postcondition of the $TicketManager$ constructor.

By the precondition of method $DispenseTicket$ and program invariant (0), we deduce that the $TicketManager$ invariant holds for **this** on entry to $DispenseTicket$. That, and the guard of the if statement, is what we need to prove that the array access $tickets[n]$ does not dereference null or index the array outside its bounds. And that is also what we need to prove the preconditions of the unpack and pack operations induced by the **expose** statement.

As this example illustrates, the $valid$ field serves not only as a bookkeeping device that keeps track of whether an object invariant can be violated, but also as an abstraction of the invariant itself, which is useful when writing specifications in the face of information hiding.

In practice, pre- and postconditions like those in Fig. 2 tend to be used in a highly stylized fashion. It is therefore possible to use good defaults when applying the methodology in practice, reducing clutter in the program text.

2. ADDITIVE FIELDS AND INVARIANTS

In this section, we amend the definition of admissible invariant to also allow object invariants that mention fields declared in superclasses (but they cannot mention the field $valid$). Thus, what we show in this section (and Section 6) make up the classic Boogie methodology [1]. However, we combine the classic Boogie methodology with the independent frames of the previous section.

We introduce a new field modifier, **additive**, which distinguishes what we shall call *additive* fields from ordinary fields. Only additive fields are allowed to be mentioned in the invariants of subclasses. The name "additive" indicates that subclasses are allowed to add constraints that guide the use of the field.

An update of an additive field f might violate not just the invariant in the class that declares f, but also the invariants in the subclasses thereof. Therefore, we define the semantics of field update for an additive field f declared in a class T to have a stronger precondition:

$o.f := E \equiv$
  assert $o \neq$ **null**;
  assert $(\forall U \bullet type(o) <: U <: T \Rightarrow (o, U).mutable)$;
  $o.f := E$

where $type(o)$ denotes the allocated type of o and $<:$ denotes the reflexive, transitive subtype relation.

Example.

Figure 3 shows a variation of the example in Fig. 2 where we have changed n to be an additive field, made the constructor postcondition more elaborate, and changed $DispenseTicket$ to be a virtual method. In Fig. 4, we show a subclass of $TicketManager$ that constrains n further by keeping fewer than 5 serviced tickets in the queue. It provides its own implementation of the virtual method $DispenseTicket$.

Note that in order to be allowed to modify $tickets$ and n, the $PruningTM$ implementation of $DispenseTicket$ must expose the object for the class frame that declares these fields, namely $TicketManager$. At the end of that **expose** statement, the $TicketManager$ invariant is checked. The $PruningTM$ invariant is checked at the end of the statement that exposes the object at $PruningTM$, namely at the end of the method. Inside the body of that **expose** statement, which includes the call to the superclass implementation of $DispenseTicket$, the $PruningTM$ invariant is allowed to be violated.

As another example of a typical use of additive fields, if the subclass invariant adds a constraint between the inherited additive field and a field declared in the subclass, as in:

```
class X extends TicketManager {
  int p;
  invariant n ⩽ p;
  ...
}
```

then a method override in the subclass may call the superclass implementation to change n and then only adjust p. In such cases, the override would only need one **expose** statement, exposing the object for X but not its superclass.

```
class TicketManager {
  int[] tickets;
  additive int n;
  invariant tickets ≠ null ∧ 0 ⩽ n ⩽ length(tickets);

  TicketManager()
    ensures n = 0 ∧ (∀ U • TicketManager <: U
        ⇒ (this, U).valid );
  {
    tickets = new int[0];
    n = 0;
  }
  ...
  virtual int DispenseTicket()
    requires ...;      // see the prose
  {
    int t = −1;
    if (n < length(tickets)) {
      t = tickets[n];
      expose (this at TicketManager) { n = n + 1; }
    }
    return t;
  }
}
```

Figure 3: A revised version of the $TicketManager$ class from Fig. 2. Here, field n is declared to be additive, which allows subclasses to further constrain n. The modifier virtual declares that the implementation of method $DispenseTicket$ varies with the allocated type of the object. An invocation of a virtual method dynamically dispatches to the implementation given in the allocated type of the object.

Method Specifications Revisited.

Let us consider the specification and verification of the classes in Figs. 3 and 4, $TicketManager$ and $PruningTM$. The constructor of $TicketManager$ starts with an implicit call to the **object** constructor, after which it gets to assume $(\mathbf{this}, \mathbf{object}).valid$. Since the field initializations establish the $TicketManager$ invariant, the constructor meets the precondition of the implicit pack operation, and thus establishes the constructor's declared postcondition.

Similarly, the $PruningTM$ constructor starts with an implicit call to the $TicketManager$ constructor, whose postcondition it then gets to assume. Since $n = 0$ satisfies the $PruningTM$ invariant, the precondition of the implicit pack operation is met, and thus the constructor postcondition is established.

The precondition of the $DispenseTicket$ method is trickier. To meet the preconditions of the unpack operation and field update in

```
class PruningTM extends TicketManager {
  invariant n < 5;

  PruningTM()
    ensures (∀ U • PruningTM <: U ⇒ (this, U).valid );
  { }
  ...
  override int DispenseTicket()
  {
    expose (this at PruningTM) {
      int t = base.DispenseTicket();
      if (5 ⩽ n) {
        int[] a = new int[length(tickets) − n];
        a[0 : length(a)] = tickets[n : length(tickets)];
        expose (this at TicketManager) {
          tickets = a; n = 0;
        }
      }
      return t;
    }
  }
}
```

Figure 4: A subclass of the Fig. 3 class $TicketManager$. Since n is declared in superclass $TicketManager$ to be additive, class $PruningTM$ is allowed to declare an invariant that further constrains n. The keyword override is used to provide a different implementation of the virtual method declared in a superclass. The keyword base is used by a method override to invoke the implementation of the method declared in the superclass. The $DispenseTicket$ override uses an array subscript of the form $[M : N]$ to indicate a sequence of indices j such that $M \leqslant j < N$.

the $TicketManager$ implementation of this method, we need the precondition:

$$\begin{array}{l}(\mathbf{this}, TicketManager).valid \ \wedge \\ (\forall\, U \bullet type(\mathbf{this}) <: U <:_{\neq} TicketManager \\ \quad \Rightarrow (\mathbf{this}, U).mutable\ )\end{array}$$

where we use $<:_{\neq}$ to denote the irreflexive, transitive subtype relation. That is, this needs to be valid for $TicketManager$ and mutable for all proper-subclass frames. However, the $PruningTM$ implementation of the method needs the precondition:

$$\begin{array}{l}(\mathbf{this}, TicketManager).valid \ \wedge\ (\mathbf{this}, PruningTM).valid \ \wedge \\ (\forall\, U \bullet type(\mathbf{this}) <: U <:_{\neq} PruningTM \\ \quad \Rightarrow (\mathbf{this}, U).mutable\ )\end{array}$$

In conclusion, we would like to use the following polymorphic precondition to specify the virtual method $DispenseTicket$:

$$\begin{array}{l}(\forall\, U \bullet \star <: U \Rightarrow (\mathbf{this}, U).valid\ ) \ \wedge \\ (\forall\, U \bullet type(\mathbf{this}) <: U <:_{\neq} \star \Rightarrow (\mathbf{this}, U).mutable\ )\end{array} \quad (1)$$

where $\star$ stands for the class where the implementation is given.

This leads to the problem of what precondition to check at calls to $DispenseTicket$. Since a call of a virtual method dynamically dispatches to the implementation given in the allocated type of the target object, the precondition to check at a dynamically dispatched call site is the polymorphic precondition with $type(\mathbf{this})$ for $\star$.

```
class Counter {
  int n;
  invariant 0 ⩽ n;

  void Inc()
    requires (∀ U • type(this) <: U ⇒ (this, U).valid );
  {
    expose (this at Counter) { n = n + 1; }
  }
}
```

Figure 5: A simple $Counter$ class with a non-virtual method to increment the counter. The class verifies under our enhanced methodology, but not under the classic Boogie methodology.

This amounts to the precondition:

$$(\forall\, U \bullet type(\mathbf{this}) <: U \Rightarrow (\mathbf{this}, U).valid\ ) \quad (2)$$

which says that all class frames of the target object are valid, that is, that the target object is consistent.

We introduce the keyword **additive** as an optional modifier for virtual methods. If supplied, we refer to the virtual method as an *additive method* and automatically supply the polymorphic precondition (1).

The soundness of this polymorphic specification approach assumes that every class provides an implementation for each of its inherited additive methods. As a programmer convenience, if no explicit override is given, the compiler supplies one automatically, in the form of a **base** call enclosed in an **expose** statement.

By marking method $DispenseTicket$ in Fig. 3 with **additive**, the code in Figs. 3 and 4 verify, as does a caller code fragment like:

```
TicketManager tm = new PruningTM();
tm.AddTicket(40);
tm.AddTicket(37);
int t = tm.DispenseTicket();
```

Note that the static type of tm is a superclass of the allocated type of the object it is holding. Nevertheless, what is known as a postcondition to the constructor is:

$$\begin{array}{l}type(tm) = PruningTM \ \wedge \\ (\forall\, U \bullet PruningTM <: U \Rightarrow (tm, U).valid\ )\end{array}$$

which implies the precondition (2) of the invocation of the virtual methods.

In the classic Boogie methodology, which also includes the polymorphic precondition (1), all fields and virtual methods were additive (and thus the word "additive" was never mentioned) [1]. In what we have presented here, we allow a mix of additive and non-additive (that is, ordinary) fields and methods.

3. BENEFITS OF NON-ADDITIVE FIELDS

The benefits of additive fields, as in the classic Boogie methodology, are clear: subclasses can further constrain the values of additive fields and can constrain the subclass fields with respect to the additive fields. Let us give some examples that show the benefits of the more liberal rules that come with non-additive fields.

3.0 Field Updates in Non-Virtual Methods

Consider the simple $Counter$ class in Fig. 5. We argue that the way it is written is the most natural way to specify and implement its non-virtual Inc method. The precondition says that the object

is consistent, which is the state of the object that typical clients will see. The fact that this precondition is so general and useful makes it a good candidate to be a default.

The class verifies under our methodology. (In fact, it would also verify with the weaker precondition $(\mathbf{this}, \mathit{Counter}).\mathit{valid}$.)

Now, suppose that its field n were additive (as it would be under the classic Boogie methodology). Then, to verify Inc, the method would need the precondition:

$$\begin{array}{l}\mathbf{requires}\ (\mathbf{this}, \mathit{Counter}).\mathit{valid} \wedge \\ \quad (\forall\ U \bullet\ \mathit{type}(\mathbf{this}) <: U <:_{\neq} \mathit{Counter} \\ \qquad \Rightarrow\ (\mathbf{this}, U).\mathit{mutable}\ );\end{array} \tag{3}$$

This precondition is awkward to establish for a client, especially if the client does not know the allocated type of the object.

The classic Boogie methodology offers two ways to solve this problem, both of which we have employed many times when verifying Spec# programs before we implemented our enhanced methodology. One solution is to make Inc into a virtual method and use the polymorphic precondition (1), which for the implementation in $\mathit{Counter}$ works out to be exactly the needed condition (3). The classic Boogie methodology was designed with this virtual-method scenario in mind. But in programming practice, always using virtual methods can seem heavy-handed. The other solution is to declare $\mathit{Counter}$ to be a sealed class, that is, one without further subclasses. This shies away from the benefits and liabilities of subtyping altogether.

In contrast, note how easily the program is specified and written when n is not additive (Fig. 5).

Before leaving this example, let us mention one other approach to coping with the additive-field update precondition in a non-virtual method (an approach discussed and rejected elsewhere [1]). One can imagine replacing the **expose** statement in the Inc method by some new operation that unpacked all of the class frames from $\mathit{Counter}$ to the allocated type of **this**. This would establish the necessary precondition for the update of the additive field n. However, the analogous operation for packing these frames is problematic for modular verification: only if the subclasses are in scope can a static verifier check that the invariants they declare hold.

3.1 Methods that Update Fields and Invoke Other Methods

As we have seen, there are two principal ways to write the precondition of a method: either the monomorphic precondition (2) or the polymorphic precondition (1). These preconditions are similar in two ways. First, either precondition boils down to that the target object of a non-**base** call must be consistent. Also, either precondition permits **base** calls: for (2), a **base** call must be done outside all **expose** statements, à la:

$$\mathbf{base}.M();\ \mathbf{expose}\ (\mathbf{this\ at}\ \mathit{Counter})\ \{\ \ldots\ \}$$

and for (1), a **base** call must be enclosed by an **expose** statement, à la:

$$\mathbf{expose}\ (\mathbf{this\ at}\ \mathit{Counter})\ \{\ \mathbf{base}.M();\ \ldots\ \}$$

The difference between (2) and (1) is that (2) permits non-**base** method calls whereas (1) permits, inside **expose** statements, updates of additive fields, but not vice versa. This is a major limitation of the classic Boogie methodology—methods must be partitioned into field-modifying methods and method-calling methods [1]. Our introduction of non-additive fields overcomes this limitation.

Let us illustrate this difference with an example. Consider the $\mathit{CoffeeTable}$ class in Fig. 6. It has virtual methods for setting the table and serving drinks, affording different $\mathit{CoffeeTable}$ subclasses the opportunity to decorate with different tablecloths and to use specialized baristas for preparing the drinks. These methods are declared additive, so that their implementations can expose the object to update additive fields. Method $\mathit{Prepare}$ sets the table and serves drinks for two, but does so only once.

```
class CoffeeTable {
  bool ready;
  void Prepare()
    requires (∀ U • type(this) <: U ⇒ (this, U).valid );
  {
    if (¬ready) {
      SetTable();
      ServeDrink(32, TEA);
      ServeDrink(37, ESPRESSO);
      expose (this at CoffeeTable) { ready = true; }
    }
  }
  additive virtual SetTable() ...
  additive virtual ServeDrink(int id, int kind) ...
}
```

Figure 6: A class whose subclasses can provide a personalized atmosphere in which to enjoy caffeinated drinks. The precondition of $\mathit{Prepare}$ is discussed in the prose.

The precondition of method $\mathit{Prepare}$ must be strong enough to let it invoke methods $\mathit{SetTable}$ and $\mathit{ServeDrink}$. Invoking these methods requires the object to be consistent, which is the precondition declared for $\mathit{Prepare}$ in Fig. 6. Since ready is a non-additive field, the enclosing **expose** statement produces a state that meets the precondition of the field update.

At the end of this **expose** statement, the (trivial) $\mathit{CoffeeTable}$ invariant is checked. Note that the field update does not violate any subclass invariants, since the definition of admissible invariant forbids subclass invariants from mentioning the non-additive field ready. It seems to make sense to let ready be non-additive, because it is local to the $\mathit{CoffeeTable}$ class and is not intended to be used in subclass invariants.

In contrast, if ready were an additive field, then it would not be possible to find a precondition for $\mathit{Prepare}$ that would allow it both to call other methods (which requires the object to be consistent) and to expose the object to modify the ready field (which, as we described in the $\mathit{Counter}$ example above, would require a precondition of the form (3)).

As a final remark about this example, one might want to make the steps performed by $\mathit{Prepare}$ specific to the $\mathit{CoffeeTable}$ subclass. For example, a subclass may set the table for more than two people. This can be achieved by declaring the $\mathit{Prepare}$ method virtual, which would also verify under our methodology using the monomorphic precondition shown in Fig. 6.

4. IMPLEMENTATION ENCODING

We have implemented our enhanced methodology in the Boogie static program verifier for Spec#. Boogie works by first translating compiled Spec# programs into the intermediate verification language BoogiePL, from which it generates verification conditions that it sends to an automatic theorem prover [0]. In its translation into BoogiePL, our implementation introduces some ghost fields that it uses to keep track of the valid/mutable state of each object. However, for performance reasons, we use a different encod-

ing than the one immediately suggested by our presentation in the previous sections. We now describe the encoding we use.

First, we introduce a notion that every mutable class frame is either *additively mutable* or *locally mutable*. To change the state of an object, we use *two* statements in the Spec# language: **additive expose** changes a valid class frame to be additively mutable for the duration of the body of the statement, and just **expose** changes a valid class frame to be locally mutable for the duration of the body. When an object is allocated, all of its class frames start off in the additively mutable state, and the end of each constructor implicitly changes the class frame to be valid.

As before, a class frame is changed to the valid state only if the object invariant declared in that class holds, and hence we maintain program invariant (0). In addition, by adding an appropriate precondition to the **additive expose** statement, we ensure that a class frame is additively mutable only if all of the subclass frames are additively mutable. Updating an additive field declared in a class T has the precondition that the T frame of the target object be additively mutable, whereas the precondition for updating a non-additive field is that the frame is either additively or locally mutable.

Second, we impose another restriction to simplify our encoding, namely that an object can be further exposed only when it has no locally mutable class frames. We encode the states of all class frames by two ghost fields, inv and $localinv$, whose values denote some superclass of the object's allocated type. For any object o and class frame T, the state of (o, T) is:

$$\begin{array}{ll} \text{valid} & \text{if } o.inv <: T \wedge o.localinv \neq base(T) \\ \text{additively mutable} & \text{if } \neg(o.inv <: T) \\ \text{locally valid} & \text{if } o.localinv = base(T) \end{array} \tag{4}$$

where $base(T)$ denotes the immediate superclass of T. Since the root class, **object**, has no fields that can be directly updated, it is also convenient to disallow the **object** class frame from being exposed. We thus have the following program invariant, for every object o:

$$type(o) <: o.inv <: \mathbf{object} \wedge (o.localinv = type(o) \vee o.inv <:_{\neq} o.localinv <: \mathbf{object})$$

At the time an object o is allocated, $o.inv = \mathbf{object} \wedge o.localinv = type(o)$. We define each of the two **expose** statements in terms of unpack and pack operations, the additive pack operation also being implicitly executed at the end of a constructor.

additive unpack o **from** $T \equiv$
  assert $o \neq \mathbf{null} \wedge o.inv = T \wedge o.localinv = type(o)$;
  $o.inv := base(T)$
additive pack o **as** $T \equiv$
  assert $o \neq \mathbf{null} \wedge o.inv = base(T) \wedge o.localinv = type(o)$;
  assert $Inv_T(o)$;
  $o.inv := T$
unpack o **from** $T \equiv$
  assert $o \neq \mathbf{null} \wedge o.inv <: T \wedge o.localinv = type(o)$;
  $o.localinv := base(T)$
pack o **as** $T \equiv$
  assert $o \neq \mathbf{null} \wedge o.inv <: T \wedge o.localinv = base(T)$;
  assert $Inv_T(o)$;
  $o.localinv := type(o)$

The semantics of field update is defined as follows, where f and g denote an additive and non-additive field, respectively, declared in a class T:

$o.f = E \equiv$
  assert $o \neq \mathbf{null} \wedge \neg(o.inv <: T)$;
  $o.f := E$
$o.g = E \equiv$
  assert $o \neq \mathbf{null}$
    $\wedge\ (o.localinv = base(T) \vee \neg(o.inv <: T))$;
  $o.g := E$

Program	LOC	#expose	#additive expose
PrettySx	424	4	0
ProverProcess	918	12	0
Tulip	≈ 2500	33	0

Figure 7: Two verified programs that use subclasses and our enhanced methodology. For each program, the columns show the number of lines of code, including Spec# contracts, but excluding comments and blank lines (except the count for Tulip); the number of expose statements; and the number of additive expose statements.

Alternative Encoding.

With the help of Ralf Sasse, we have experimented with one alternative encoding. The alternative encoding stays closer to our description in the earlier part of the paper, using the ghost field inv as above and using a ghost field $validFor$ whose value is a set of classes, representing those class frames for which the object is valid. Under this encoding, we do not need the second restriction that we imposed above. Alas, we found that the alternative encoding degraded theorem-proving performance rather than improving it as we had hoped.

Run-Time Behavior.

In this paper, we focus on specifications and static verification, but the checks we prescribe can also be performed dynamically. Indeed, Spec# performs a subset of these checks at run time. In particular, the Spec# compiler adds one bit to every class, namely the boolean $valid$ as described in our previous sections. (The ownership system that we describe in Section 6 is not represented at run time.) Thus, the run-time representation used in Spec# for our enhanced methodology is the same as when Spec# supported only the classic Boogie methodology.

5. EXPERIENCE

We have found our enhanced methodology to be widely applicable in programming practice. Since we implemented it in Boogie—now more than 12 months ago—we have found that we use non-additive fields almost exclusively. The enhanced methodology makes a noticeable difference in the ease with which programs are architected and specified.

Three programs that are specified and verified with the enhanced methodology are reported in Fig. 7. *PrettySx* is a program for parsing, refactoring, and pretty printing S-expressions, *ProverProcess* is the part of Boogie itself that communicates with the underlying theorem provers, and Tulip is an operating-system driver. All of these programs make use of class hierarchies and subclasses. The programs make use of **expose** statements, but never any **additive expose** statements, because there are no additive fields.

In *PrettySx*, a method *ReadToken* contains both field updates

```
class LuxuryCar extends Car {
  Radio r;
  invariant r ≠ null
    ∧ (100 ⩽ speed ⇒ r.soundBoosterSetting = MAX);
  ...
}
```

Figure 8: A variation of the $LuxuryCar$ class in Fig. 0. The invariant says that for sufficiently high speeds, the radio's sound booster is in the highest setting.

(inside an **expose** statement) and a call to $PeekToken$, which also contains field updates. These methods could be used as public entry points of the class, but doing so under the classic Boogie methodology would require making the two methods virtual and putting the body of $PeekToken$ into a worker routine that is called from it and $ReadToken$, unlike the current design where $ReadToken$ calls $PeekToken$ directly.

The restrictions described in Section 4 that we imposed to enable our current implementation encoding—introducing two flavors of mutability that are obtained by different statements, and allowing only one of an object's class frames to be locally mutable at any one time—have not been confining, even noticed, in practice.

The first attempt to verify the $ProverProcess$ class hierarchy was done under the classic Boogie methodology, that is, by treating all fields as being additive. This led to a number of complications where it was not clear how to proceed. In contrast, under the enhanced methodology, specification and verification was straightforward.

6. AGGREGATE OBJECTS

One object is often implemented in terms of other objects, all together making up a logical *aggregate object*. Invariants often span the individual objects in an aggregate, which comes down to writing invariants that mention fields of fields. An example is the class of radio-equipped luxury cars in Fig. 8. The definition of admissible invariant can be changed to allow such invariants, but doing so requires introducing some rules that constrain the structure of the heap. Without such rules, then in a module where class $Radio$ is in scope but $LuxuryCar$ is not, a program could turn off $soundBoosterSetting$ for a radio object, even if that radio were used in a fast-traveling luxury car, thus violating the $LuxuryCar$ invariant.

A useful heap structuring technique is *ownership* [7, 6, 18, 5], which has been applied in the verification of object-oriented programs [17, 13, 9]. Ownership is mostly orthogonal to the issue we address in this paper, but we include a treatment of it to show how our formalization incorporates ownership. In our setting, an *owner* is an (object reference, class frame) pair [13], which we encode by adding a field $owner$ to every object. The value of the $owner$ field is either a pair or the special value $\bot$, which indicates that the object has no owner.

The crucial property we aim to achieve with ownership is the following program invariant [1]:

$$(\forall\, o, T \bullet\ type(o) <:\ T \wedge (o, T).mutable \wedge o.owner \neq \bot \Rightarrow\ o.owner.mutable\ ) \tag{5}$$

where o ranges over non-null, allocated objects and T ranges over class names. This property says that an object can have a mutable frame only if the owner is mutable. Stated in the contrapositive, if a frame of an object is valid, then so are all the frames of all the objects it owns.

To maintain program invariant (5), we adjust the semantics of the unpack and pack operations to consider ownership:

```
unpack o from T ≡
  assert o ≠ null ∧ (o, T).valid;
  assert o.owner = ⊥ ∨ o.owner.mutable;
  (o, T).valid := false
pack o as T ≡
  assert o ≠ null ∧ (o, T).mutable;
  assert (∀ r • r.owner = (o, T) ⇒ (∀ R • (r, R).valid ));
  assert Inv_T(o);
  (o, T).valid := true
```

The unpack operation checks that the owner is already mutable, and the pack operation checks that all owned objects are valid. If an object has an owner in the valid state, we say that the object is *committed* [1]. The precondition given by the second assertion in the semantics of unpack thus says that o is not committed.

Every method implementation that updates a field $o.f$ and exposes the object o must meet the precondition of the unpack operation, and in particular must show that o is not committed. In most cases, all reference-valued parameters (including the receiver parameter, **this**) and return values are specified to be not committed (an exception is the specification of pure methods [8]). So, for the purposes of this paper, we implicitly apply pre- and postconditions that specify this (which makes all examples shown earlier in the paper apply also in the presence of aggregate objects).

When an object is allocated, its $owner$ starts off being $\bot$. We allow programs to assign the $owner$ field, which has the following semantics:

```
o.owner = ⊥ ≡
  assert o ≠ null ∧ (o.owner = ⊥ ∨ o.owner.mutable);
  o.owner := ⊥
o.owner = (ow, T) ≡
  assert o ≠ null ∧ (o.owner = ⊥ ∨ o.owner.mutable);
  assert ow ≠ null ∧ type(ow) <: T ∧ (ow, T).mutable;
  o.owner := (ow, T)
```

These rules say that an $owner$ assignment (ownership transfer) is allowed only when the old and new owners are mutable.

Representation Fields.

With the ownership system in place, and in particular with program invariant (5), we can now allow fields of owned objects to be mentioned in invariants. By our rules, a field $o.f$ is allowed to be modified only if o is mutable; by program invariant (5), we then have that $o.f$ is modified only at times when $o.owner$ is mutable. Therefore, if $o.owner = (ow, T)$, it is sound to allow the T invariant of object ow to depend on $o.f$.

Following standard practice, we define admissible invariant in a way that can be enforced syntactically: we introduce field modifier **rep** (for *representation*), which can be applied to reference-valued fields. If a class T declares a **rep** field f, then we allow the invariant in class T (but not in proper subclasses of T, even if f is additive) to dereference f. Moreover, such a **rep** declaration gives rise to the following implicit object invariant:

$$\textbf{invariant}\ f = \textbf{null}\ \vee\ f.owner = (\textbf{this}, T); \tag{6}$$

The $owner$ field is not allowed to be mentioned explicitly in object invariants.

```
class Car {
  additive int speed;
  additive virtual void SetSpeed(int kmph)
  ...
}
class LuxuryCar extends Car {
  rep Radio r;
  invariant r ≠ null
      ∧ (100 ⩽ speed ⇒ r.soundBoosterSetting = MAX);
  override void SetSpeed(int kmph) {
    expose (this at LuxuryCar) {
      base.SetSpeed(kmph);
      if (100 ⩽ speed) { r.EngageBooster(MAX); }
    }
  }
  ...
}
class Radio {
  int soundBoosterSetting;
  void EngageBooster(int b)
    requires (∀ U • type(this) <: U ⇒ (this, U).valid );
    ensures soundBoosterSetting = b;
  ...
}
```

Figure 9: The Car and $LuxuryCar$ example where the $LuxuryCar$ includes a $Radio$ representation object and overrides $SetSpeed$ to maintain its invariant about the radio. The field modifier rep indicates that r is a representation field; this allows $r.soundBoosterSetting$ to be mentioned in the invariant and lets $SetSpeed$ meet the precondition of its call to $EngageBooster$.

Example.

Continuing the example in Fig. 8, we show the $LuxuryCar$ implementation of $SetSpeed$ in Fig. 9. The invariant in $LuxuryCar$ is admissible: it is allowed to mention the superclass field $speed$, since $speed$ is declared to be additive, and it is allowed to dereference r, since r is declared to be a representation field.

For the verification of the $LuxuryCar$ implementation of the additive method $SetSpeed$, the polymorphic precondition for additive methods and the implicit not-committed precondition of all parameters imply the precondition of the unpack operation. With the effect of the unpack operation, the precondition of the call to the Car implementation of $SetSpeed$ is met. The precondition of the call to $r.EngageBooster$ requires r to be not committed and all of r's class frames to be valid. The first part of this precondition is met on account of that r has a mutable owner: the implicit **rep**-field invariant (6) says $r.owner$ is $(\mathbf{this}, LuxuryCar)$, which is mutable at the time of the call. The second part of the precondition is met on account of that $(\mathbf{this}, LuxuryCar)$ is valid on entry to the **expose** statement, which by program invariant (5) implies that all class frames of owned objects are valid, and the fact that $valid$ changes only for $(\mathbf{this}, LuxuryCar)$ between the entry of the **expose** statement and the call to $EngageBooster$.

Note that without the **rep** declaration and program invariant (5), it would not have been possible to prove the precondition of the call to $EngageBooster$.

Implementation Encoding.

Boogie implements the two components of the pair-valued field $owner$ as separate fields, $ownerRef$ and $ownerFrame$. From that and the encoding (4), the encoding of what we have mentioned in this section follows straightforwardly. For example, the condition $o.owner.valid$ is encoded as:

$$o.ownerRef.inv <: o.ownerFrame \;\wedge\; o.ownerRef.localinv \neq base(o.ownerFrame)$$

and the second assertion in the semantics of the pack operation in this section is encoded as:

$$(\forall\, r \bullet\; r.ownerRef = o \wedge r.ownerFrame = T \;\Rightarrow\; r.inv = type(r) \wedge r.localinv = type(r)\;)$$

The condition we described in this section as $o.owner = \bot$ is encoded in our implementation as $o.ownerFrame = \bot$. (When $o.ownerFrame = \bot$, our implementation still treats the value of $o.ownerRef$ as significant, but for reasons that are not directly related to this paper.)

The soundness of our methodology depends on that a **rep** field declared in a class T can be dereferenced only in the invariants of class T, not in the invariants of proper subclasses of T. In our implementation, we enforce this restriction by disallowing fields that are both **rep** and **additive**.

7. RELATED WORK

Another approach to modularly verifying object-oriented program is based on universe types by Müller *et al.* [17, 19]. That approach does not tease out the various class frames of objects and does not let different class frames guard their own universes of representation objects. Instead, object invariants apply to entire objects and owners are single objects.

In the encoding of Jacobs and Piessens [10], a subclass frame owns its superclass frame. This shifts the subclass problem into the ownership problem. It is not clear how to adapt our methodology in that setting, because it would need a weaker version of program invariant (5), where one class frame could be valid even when a superclass frame that it owns is not. Solving this problem could give a more general solution that allows arbitrary parts of aggregate objects temporarily to violate their invariants.

There may be some hope for such a more general solution, because a situation of a similar nature exists for class invariants (that is, invariants that govern the data common to all instances of a class). Leino and Müller allow a class to be exposed even if not all the classes that depend on it have been exposed [14]. This is achieved by distinguishing between a class being valid and a class being *transitively* valid, meaning that it and all its transitive dependees are valid.

Verification of object-oriented program has also been studied in the context of separation logic [20], but the work is more similar to the classic Boogie methodology in its treatment of subclasses.

Dynamic frames [12] are a general and promising specification technique that handles abstraction and information hiding in the heap. To use the technique for examples like ours, one would need to develop a suitable specification idiom that would take the place of our methodology.

Throughout the paper, we have mentioned the Boogie methodology. Of the work we have mentioned in this section, the Boogie methodology is the one with the most elaborate implementation in a program verifier. Extensions of the Boogie methodology have been developed, for example for visibility-based invariants [13, 3] and concurrency [9, 11].

Another variation on the Boogie methodology, where violated invariants have to be listed as exclusions, is investigated by Middelkoop *et al.* [16]. It has not yet been applied in the context of ag-

gregate objects, but it would be interesting to explore how it might be used with the non-additive fields in our work.

A software design issue similar to the one we have described arises in the context of concurrent programs with non-reentrant monitors (see, for example, [4]). Instead of deciding the placement of **expose** statements and using appropriate methods preconditions, concurrent programming involves the decision of where to place lock statements in order to avoid race conditions and deadlocks. The placement of lock statements thus affects method preconditions and sometimes involves the use of worker routines.

8. CONCLUSIONS

The extensibility of well-designed object-oriented classes can make writing large programs more manageable. However, extensibility comes at a price and is not always used. In this paper, we have presented a methodology for specifying and verifying programs where programmers can decide when they want to make use of the extensibility and when they want to limit it. The methodology is a refinement of the Boogie methodology for object invariants. We have implemented the methodology in the Boogie program verifier for Spec#, and we have found that the additional flexibility in the enhanced methodology makes a noticeable difference when specifying and writing programs to be verified.

In our experience, we have not come across the need for an object to have more than one locally mutable class frame at a time. Therefore, it seems realistic to impose that restriction, which we have done in our implementation and which for us has resulted in improved theorem-prover performance.

A remaining challenge is to overcome our methodology's limitation that a field cannot be both **rep** and **additive**, which could potentially be useful in some programs. Since the encoding of the methodology affects the correctness checks performed throughout the program (for example as preconditions of method calls and field updates), another improvement we would like to see is some alternative encoding of our methodology that would streamline theorem-prover performance.

Acknowledgments.

We are grateful to the rest of the Spec# team for many valuable discussions, to Ralf Sasse for implementing the alternative encoding, to Kevin Bierhoff for specifying and verifying Tulip, and to the users of Spec# who previously had expressed their frustrations with the rigidity of the classic Boogie methodology.

9. REFERENCES

[0] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, 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.

[1] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[2] 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.

[3] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, volume 3125 of *LNCS*, pages 54–84. Springer, July 2004.

[4] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, DEC SRC, January 1989.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin 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.

[6] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[7] 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.

[8] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE 2007*, volume 4422 of *LNCS*, pages 336–351. Springer, March 2007.

[9] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM 2005*, pages 137–146. IEEE, September 2005.

[10] Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In *FTfJP 2006*, July 2006.

[11] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *ICFEM 2006*, volume 4260 of *LNCS*, pages 420–439. Springer, November 2006.

[12] 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.

[13] 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.

[14] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *FM 2005*, volume 3582 of *LNCS*, pages 26–42. Springer, July 2005.

[15] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP 2007*, volume 4421 of *LNCS*, pages 80–94. Springer, March 2007.

[16] 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.

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

[18] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.

[19] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[20] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL 2005*, pages 247–258. ACM, January 2005.