Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 195 (2008) 211–229
www.elsevier.com/locate/entcs

Invariants for Non-Hierarchical Object Structures

Ronald Middelkoop Cornelis Huizing Ruurd Kuiper
Erik J. Luit

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{*r.middelkoop,c.huizing,r.kuiper,e.j.luit*}*@tue.nl*

Abstract

We present a Hoare-style specification and verification approach for invariants in sequential OO programs. It allows invariants over non-hierarchical object structures, in which update patterns that span several objects and methods occur frequently. This gives rise to invalidating and subsequent re-establishing of invariants in a way that compromises standard data induction, which assumes invariants hold when a method is called. We provide specification constructs (inc and coop) that identify objects and methods involved in such patterns, allowing a refined form of data induction. The approach now handles practical designs, as illustrated by a specification of the Observer Pattern.

Keywords: Invariants, Formal specification, Program verification, Object-oriented programs

1 Introduction

Traditionally, an invariant is a consistency property of the data of a single object, enabling reduced specification effort. Data induction is, essentially, the observation that if an object is only approached through its methods, a property is invariant if it is established by the constructors and preserved by the methods [7]. But in practice, invariants may range over more than one object. Furthermore, an invariant is sometimes invalid at a method call, in particular, when this method is called to re-establish the invariant. Obviously this method can not rely on the invariant. Therefore, data induction must be refined. Some approaches successfully exploit the dependency hierarchy between objects [10,6]. However, there are natural OO designs that are inherently non-hierarchical. A case in point is the Observer Pattern [3]. The approach in [2] allows for non-hierarchical invariants, but drops data induction. We present an alternative that retains it.

First, we introduce the specification construct **inc** that makes explicit that a method preserves, but does not rely on, certain invariants of certain objects. We

doi:10.1016/j.entcs.2007.08.034

extend results from [9]: instead of the previously used fixed set of object references, predicates are introduced to describe a set of objects involved. We argue that the additional flexibility offered by **inc** is essential in the use of invariants over non-hierarchical object structures. Second, we introduce the **coop** construct that specifies which invariants might be invalidated when a field is assigned to. This enables verification of invariants even when their definition is not visible. In particular, this supports modular development. We extend previous results with predicates to describe the set of objects involved. Third, we remove a limitation on method calls in while and if statements. Finally, the consequences of these extensions are incorporated in a proof system. More invariants are admissible and more implementations can be verified than before. In fact, whereas the approach previously could only be used for somewhat tailor-made examples, the extensions enable to specify the inspiration for the approach: the Observer Pattern.

Following this introduction, section 2 introduces invariants. Section 3 introduces the **inc** construct, section 4 introduces the **coop** construct and section 5 contains the formalization. Section 6 describes related and future work. Section 7 concludes the paper.

2 Invariants in OO development

OO programs are structured by a decomposition into classes, which group related data and methods operating on this data. A method of one class can use (objects of) another class in its implementation. A *proper user* of a class C is a method that does not contain references to fields defined in C, but only interacts with objects of class C via C's methods. Note that our proof technique does not require restriction to proper use.

We say a method M *preserves* a property if, when the property holds when M is called, it also holds when M terminates. An *invariant property* of an object is established by the object's constructor and preserved by all the object's methods. For every Book object in the (Java-like) example in Figure 1, "title is not null" is an invariant property. Proper users of Book do not invalidate the invariant property of a Book object. For every Book object, once the invariant property is established by the constructor, it always holds. The program capitalizes on this. Book's hasTitle doesn't check if the title is non-null. UI's method showHasTitle appends to the result of getTitle without checking if it is null.

A Hoare-style method specification contains a pre- and a post-condition in terms of the data of its class. For each method M, it should be verified that it terminates normally and that its postcondition holds (we do not consider exception handling). A verifier can assume that the precondition holds when M is called. When verification of M fails without the assumption that a certain property holds when M is called we say (the verification of) M *relies on* that property. In Figure 1, Book's method hasTitle and UI's method showTitle rely on the invariant property of Books this and b. If a property is specified as a precondition, a user (like method showHasTitle) must prove that the property holds before a call. Thus any such user

```
class Book {
  String title;
  Book() { this.title := "unknown"; }
  boolean hasTitle(String t) {
    return this.title.equals(t);
  }
  int getTitle() { return this.title; }
  void setTitle(String newT) {
    if (newT != null) { this.title := newT; };
  }
}
```

```
class UI {
  void showHasTitle(Book b, String t) {
    if (b != null) {
      boolean ht := b.hasTitle(t);
      //show boolean ht on the screen
    }
  void showTitle(Book b) {
    if (b != null) {
      String s := b.getTitle();
      String s := s.concat(" is the title"))};
      //show String s on the screen
    }
}
```

Fig. 1. Book/UI, invariant properties and their use

relies on the property as well. Specification of a property in the precondition of these users means *their* users must prove the property holds before a call, and so on. The property propagates throughout the program's specification. An invariant property can be specified with an *invariant*. Consider the following aim:

Aim: *The verifier of a method M that relies on an invariant I can*
1) assume that I holds when M is called, and
2) deduce if a method called by M preserves I.

When the aim is met, propagation of invariant properties is prevented, significantly reducing specification overhead. Furthermore, the code is more flexible, as a re-implementation of a method can rely on a different set of invariants without affecting users. Besides these advantages, invariants allow the specification of data consistency properties and behavioral properties to be separate concerns. This makes communication of such properties much easier [8]. Finally, they support the specification of a class in terms of an abstraction of its data [4] (see section 4).

The Book/UI example suggests that the assumptions in the aim above are sound when an invariant is 1) established by the constructor of an object and 2) preserved by every method in the program. Due to what is known as the *call-back problem*, this is not the case. The call-back problem is illustrated in Figure 2. As the example shows, an invariant is specified as a predicate on the logical variable **this**, that represents the object the invariants applies to. Assume that method calcVal always returns a value that is greater than this.i. Assume that the constructor of a C object establishes its invariant and that every other method in the program (including calcVal and m) preserves it. Then the invariant of C object this still might not hold when the second call to calcVal in m is made (as the assignment to this.i might invalidate it).

More generally, a method M may temporarily invalidate an invariant. When M calls another method before the invariant is re-established, a method that relies on the invariant might be called while the invariant does not hold. The most straightforward solution to the call-back problem is to require that any invariant that is invalidated by a method is re-established before a method call is made. These observations lead to the following theorem, whose conclusion clearly meets the aim.

```
class C {
  int i,j;
  inv this.i < this.j;
  int m() { this.i := this.calcVal(); this.j := this.calcVal(); } //constructor and calcVal omitted
}
```

Fig. 2. the call-back problem

Theorem 2.1 (data induction) *If, for any invariant I of any object,*

1) the constructor of that object establishes I, and

2) all methods in the program preserve I, and

3) no method is called while I is invalid

Then, for any method M, for any invariant I of any object,

1) unless M is the constructor of that object, I holds when M is called, and

2) I holds when a method called by M terminates

Proof. Proof (by induction on the length of execution sequences) is straightforward □

Execution of a program that has been proven correct with the *classical technique* meets the premises of Theorem 2.1. In the classical technique, described and proven sound in [11], only *local invariants* are admissible. A *local invariant* is an invariant that only depends on the fields of the object it applies to (i.e., the predicate that defines it only contains references of the form **this**.f).

3 Non-local Invariants

3.1 The specification construct inc

We call a specification *feasible* when there is an implementation of this specification that can be verified. This section shows that many natural OO designs that include non-local invariants are infeasible due to the third premise of Theorem 2.1. The specification construct **inc** is presented as a solution to this problem. It allows a method M to specify that M preserves, but does not rely on certain invariants of certain objects. It is also argued that many non-hierarchical designs are *only* feasible in a specification language that includes a construct like **inc**. There is a hierarchy between two objects if, when a method M is called on one, no method can be called (or field accessed) on the other until M terminates.

Non-local invariants are natural in many OO designs. This is illustrated by Figure 3, which could be part of a library management system. The invariants are named to allow one to distinguish between different invariants of a class. We ignore the orthogonal issue of how to specify what a method leaves untouched [5,10,16]. This problem is alleviated, but not solved by invariants. We assume that relevant changes are reflected in the method's postcondition. Due to the third premise of Theorem 2.1, the design in Figure 3 is infeasible (given proper use). The assignment to loaned invalidates the invariant of Member this. To re-establish the invariant, field

```
class Member {
  Book loaned;

  inv MI def this.loaned ≠ null ⇒
              this.loaned.loanedTo = this;

  void loan(Book b) {
    pre:  this.loaned = null ∧
          b.loanedTo = null;
    post: this.loaned = b;
    impl: this.loaned := b; b.loanTo(this);
  } //other fields and methods omitted
}
```

```
class Book {
  Member loanedTo;

  inv BI def this.loanedTo ≠ null ⇒
            this.loanedTo.loaned = this;

  void loanTo(Member m) {
    inc:  MI(m);
    pre:  this.loanedTo = null ∧ m.loaned = this;
    post: this.loanedTo = m;
    impl: this.loanedTo := m;
  } //other fields and methods omitted
}
```

Fig. 3. Example of a non-local invariant

loanedTo of Book b needs to be updated. This is not possible without a method call. Book provides method loanTo for this purpose. However, the invariant needs to be re-established before the method call that re-establishes it is allowed! Updating loanedTo before loaned is similarly impossible. The essence of the problem is that no single method can update all relevant fields when re-establishing an invariant.

The specification construct **inc** (for inconsistent), first introduced in [9], offers flexibility at little cost. In this particular example, the specification of method loanTo includes **inc:** MI(m). This makes explicit that (the verification of) loanTo does not rely on invariant MI of parameter m. This allows loanTo to be called by method loan after the assignment to this.b. Method loanTo does not have to re-establish the invariant (but from its postcondition it can be deduced that it does).

More generally, with every method M, a so called *inc-set* is associated. The inc-set is specified by the **inc** construct. By default, the set is empty. In the approach introduced in [9] the inc-set can only be specified as a fixed set of *reference invariants* $I(r)$. This approach is generalized here. The inc-set of a method M is a set of elements (C, I, P), with C a classname, I the name of an invariant specified in class C and P a predicate. References in P start with either a method parameter (for instance, this) or the logical variable **inc**. The meaning is that for any object **inc** of class C such that P holds when M is called, $I(\mathbf{inc})$ is preserved, but not relied upon by method M. An element $I(r)$ is shorthand for the element $(C, I, \mathbf{inc} = r)$, where C is the class that defines invariant I referred to by r. Theorem 3.1 reflects the addition of the **inc** construct.

Theorem 3.1 (data induction with inc) *If, for any invariant I of any object,*

1) *the constructor of that object establishes I, and*
2) *all methods in the program preserve I, and*
3) *while I is invalid, any method that is called specifies that it doesn't rely on I*

Then, for any method M, for any invariant I of any object,

1) *unless M is the constructor of that object or specifies that it does not rely on I, I holds when M is called and*
2) *unless I is invalid when a method M' is called by M, I holds when M' terminates*

Proof. Proof (by induction on the length of execution sequences) is straightforward □

The premise of Theorem 3.1 is weaker than that of Theorem 2.1. For any method M, for any invariant I, the conclusion is weaker only in two cases: 1) M calls a method while I is invalid. However, given such a call, premise 3 of Theorem 2.1 is not met and Theorem 2.1 cannot be applied. 2) M specifies that it does not rely on I. In that case, I cannot be assumed to hold when M is called. However, the choice to include an invariant in the inc-set of M is made by M's developer. In Figure 4, the **inc** construct is applied to a more complex program. It shows how non-local invariants can be verified in a setting without information hiding. This example is derived from the Observer Pattern [3]. Users of a CSubject object can set a value, here **int** d, with method setD. A CObserver object has a field cs that refers to a CSubject object. We say it *observes* that CSubject. Users of a CObserver can retrieve a value derived from the observed CObserver's field d by calling method getVal. This value is represented by $f(this.cs.d)$ (that is, $f(this.cs.d)$ is a placeholder for an integer expression that depends on $this.cs.d$). The most straightforward way to implement getVal is to calculate $f(this.cs.d)$ every time getVal is called. However, assume that retrievals of $f(this.cs.d)$ are more frequent than changes to d, and assume that it is relatively expensive to calculate $f(this.cs.d)$. Then it is more efficient to store $f(this.cs.d)$ in a variable that is updated when d is updated. Also, this variable is returned when $f(this.cs.d)$ is requested. This implementation is specified in Figure 4. Note that a reference invariant $I(r)$, where I is the name of an invariant defined in (a supertype of) the static type of reference r, may occur in a predicate. Reference invariant $I(r)$ holds iff $r = null$ holds or $P[r/\mathbf{this}]$ holds, where P is the predicate identified by $I(r)$ and $P[r/\mathbf{this}]$ is the capture-avoiding substitution of **this** by r in predicate P.

To apply Theorem 3.1, every method must preserve all invariants. Consider an arbitrary statement in a method M. Given non-local invariants without restrictions, this statement can invalidate an arbitrary invariant of an arbitrary object. Preservation (of all invariants) is guaranteed when the verifier of M proves, for any class C, for any invariant I defined in C, for an arbitrary object of class C, that invariant I of that object 1) cannot be invalidated by the statement or 2) is re-established before the end of M. Perhaps surprisingly, such a proof is often straightforward. For most invariants, it can be determined statically that they cannot be invalidated by the statement (for instance, an assignment to a field that is not involved in the invariant). Otherwise, a more elaborate proof is needed. For example, method setD in Figure 4 contains an assignment to field d of class CSubject. A reference to such a field also occurs in invariant I of class CObserver. To deduce that only CObserver this.co can be invalid, the proof has to use *flanking* invariant J (we call J a flanking invariant because no method relies on J when invariant I is removed). To re-establish the invariant, this.co.update() is called. This call is allowed as update specifies that it does not rely on invariant I of the object on which it is called. Note that the call from update to getD is allowed due to the inc-set of getD and the validity of flanking invariant J in the prestate of update.

```
class CSubject {
  int d;
  CObserver co;

  public void setD(int newD) {
    post: this.d = newD;
    impl: this.d := newD;
          if (this.co != null)
            { this.co.update(); }
  }

  public int getD() {
    inc:  I(o);
    post: result = this.d;
    impl: return this.d;
  }

  public void attach(CObserver o) {
    inc:  I(o), J(o);
    pre:  this.co = null ∧ o.cs = this;
    post: I(o) ∧ J(o);
    impl: this.co := o; o.update(this.d);
  }
}
```

```
class CObserver {
  CSubject cs;
  int i;

  inv I def this.i = f(this.cs.d);
  inv J def this = this.cs.co;

  public CObserver(CSubject toObs) {
    pre:  toObs.co = null;
    post: this.cs = toObs;
    impl: this.cs := toObs; toObs.attach(this);
  }

  public int getVal() {
    post: result = f(this.cs.d);
    impl: return this.i;
  }

  void update() {
    inc:  I(this);
    post: I(this);
    impl: this.i := f(this.cs.getD());
  }
}
```

Fig. 4. Observer Pattern, single observer: example of inc-sets

We argue that a construct that specifies that a method does not rely on certain invariants is essential. On the one hand, we have the examples in Figures 3 and 4. These show natural OO designs that cannot be implemented without a call to a method that has a reference to an object with an invalid invariant. On the other hand, we have Figure 1, which shows an equally natural design in which methods implicitly rely on invariants of objects they have a reference to. When this is disallowed, there is an unwanted propagation of properties throughout the specification. *Only* given a construct that makes explicit that a method does not rely on certain invariants are both designs possible.

3.2 Generalized inc-sets

The generalization of inc-sets allows additional restrictions on the conditions under which an invariant is not relied upon to be specified conveniently. For instance, $(C, I, \mathbf{inc} = this.s \wedge (\mathbf{inc}.a = 4 \vee this.a = 4))$ specifies that a method does not rely on invariant I of object this.s when either field a of this.s of or field a of object this has value 4.

More important, however, is that the generalized notation does not limit the inc-set to a fixed set of reference invariants. For instance, **inc:** (CObserver, I, *true*) specifies that a method does not rely on invariant I of *any* object **inc** of class CObserver. The example in Figure 5 capitalizes on this increased expressivity. This design is not feasible without generalized inc-sets. In this example, invariant J contains a reference of the form $r.f^i$, where $i \geq 0$. Such a reference $r.f^i$ represents reference r followed by i applications of field access $.f$. Invariant J specifies that a CObserver occurs in the list of CObservers maintained by the CSubject it observes.

```
class CSubject {
  int d;
  ONode on;

  public void setD(int newD) {
    post: this.d = newD;
    impl: this.d := newD; ONode iter := on;
          while (iter != null) {
              iter.obs.update(newD);
              iter := iter.next; }
  }

  public int getD() {
    inc:  (Observer, I, inc.cs = this);
    post: result = this.d;
    impl: return this.d;
  }

  public void attach(CObserver o) {
    inc:  I(o), J(o);
    pre:  o.cs = this;
    post: I(o) ∧ J(o);
    impl: ONode n := new ONode(o, on);
          on := n; o.update(this.d);
  }
}
class ONode {
  Observer obs;
  ONode next;

  inv I def this.obs ≠ null;

  public ONode(Observer o, ONode n) {
    inc:  I(o), J(o)
    post: this.obs = o ∧ this.next = n;
    impl: this.obs := o; this.next := n;
  }
}
```

```
class CObserver {
  CSubject cs;
  int i;

  inv I def this.i = f(this.cs.d);
  inv J def ∃i • this = this.cs.on.next^i.obs;

  public CObserver(CSubject toObs) {
    pre:  toObs.co = null;
    post: this.cs = toObs;
    impl: this.cs := toObs; toObs.attach(this);
  }

  public int getVal() {
    post: result = this.i;
    impl: return this.i;
  }

  void update() {
    inc:  (Observer, I, inc.cs = this.cs);
    post: I(this);
    impl: this.i := f(this.cs.getD());
  }
}
```

Fig. 5. Observer Pattern, multiple observers: example of generalized inc-sets

The call to the update method in setD is allowed as update specifies that it does not rely on invariant I of any object **inc** of class CObserver that observes the same CSubject as the object that update is called on. This set of invariants cannot be specified as a fixed set of reference invariants. Note that update preserves all invariants, even those specified in its inc-set. This allows the verifier of setD to conclude that invariants of objects that have been updated already are preserved by an update call. setD is not a proper user of ONode. This could be modified.

4 Information Hiding

4.1 The specification construct coop

This section presents the specification construct *coop*, first introduced in [9]. This construct specifies which invariants might be invalidated when a field is assigned to. This allows the verification of designs that include non-local invariants, even when

these invariants can be hidden from a class.

So far, an important concept in OO verification has been ignored, namely that of *information hiding*. Information hiding [13] is an important OO design principle. Design decisions that are not relevant to a user should be hidden from that user (by means of abstraction). These decisions can then be changed without affecting that user. With *modular development* [10] of (a method of) a class C we mean that all except a finite and explicit set of classes are hidden from the developer of (this method of) C. The benefit (of modular development) is that changes to hidden classes, or the addition of new classes, do not affect the verification of (this method of) C. Given a mechanism that can hide an invariant from a class, most OO designs are infeasible unless invariants are restricted. Due to the second premise of Theorem 3.1, the verifier must prove, for an arbitrary hidden invariant I of an arbitrary object of an arbitrary class, that I 1) cannot be invalidated by the statement or 2) is re-established before the end of M.

Every execution of a program that has been proven correct with the technique that we introduce in section 5 meets the premises of Theorem 3.1, even when invariants can be hidden from a class. To this end, the technique 1) restricts invariants so that they can only be invalidated by assignment statements 2) has an admissibility obligation on invariants that uses the specification construct **coop**. Figure 6 illustrates the intuition behind this construct. An assignment to field i of a CObserver object can invalidate invariant I of that object. As field i is specified as **int** i **coop** I(**this**), a verifier can assume that that invariant is the *only* invariant that might be invalidated.

More generally, a so called *coop-set* specified by the **coop** construct is associated with every field f. By default, the coop-set is empty. We generalize the approach in [9]. A coop-set associated with a field f is a set of elements (C, I, P), with C a classname, I the name of an invariant specified in class C and P a predicate in which all references consist of one of the keyword logical variables **this** or **dep**, followed by zero or more field accesses. We say field f *cooperates with* invariant I of any object **dep** of class C for which P holds at the time field f is assigned to. The admissibility obligation guarantees the following property. When a field f of an object is assigned to, any invariant *not* cooperated with by f is *not* invalidated by the assignment. An element $I(r)$ is shorthand for the element $(C, I, \mathbf{dep} = r)$, where C is the class that defines invariant I referred to by r (i.e, it specifies that the field cooperates with invariant I of object r, when such an object exists).

Figure 6 shows the potential of this solution. One goal of the Observer Pattern is 'loose coupling' between CSubject and CObserver [3]: class CObserver should be hidden from class CSubject. Then, changing CObserver does not affect CSubject. In particular, this allows objects of different classes to observe a CSubject. The Observer pattern uses abstraction to allow this hiding. Class Subject is an abstraction of all classes that can be observed. Likewise, class Observer is an abstraction of all classes that can observe a Subject. Class CObserver is hidden from classes CSubject and Subject, and class CSubject is hidden from class Observer. The main problem is that different implementations of Observer have different invariants, which are hid-

```
class Subject {
  Observer o coop I(this.o), J(this.o);

  public void attach(Observer o) {
    inc:  I(o), J(o);
    pre:  this.o = null ∧ o.s = this;
    post: I(o) ∧ J(o);
    impl: this.o := o; o.update(this.d);
  }
}

class CSubject extends Subject {
  int d coop I(this.o);

  public void setD(int newD) {
    post: this.d = newD;
    impl: this.d := newD;
          if (this.o != null) { this.o.update(newD);}
  }

  public int getD() {
    inc:  I(o);
    post: result = this.d;
    impl: return this.d;
  }
}
```

```
interface Observer {
  abstract Subject s coop J(this);

  abstract inv I;
  inv J def this = this.s.o;

  void update(int d) {
    inc:  I(this);
    post: I(this);
  }
}

class CObserver implements Observer{
  int i coop I(this);
  CSubject cs coop I(this);

  def s by this.cs;

  def I by this.i = f(this.cs.d) ∧ J(this);

  public CObserver(CSubject toObs) {
    pre:  toObs.o = null;
    post: this.cs = toObs;
    impl: this.cs := toObs; toObs.attach(this);
  }

  void update(int d) {
    impl: this.i := f(this.cs.getD());
  }
}
```

Fig. 6. Observer Pattern with information hiding and coop-sets

den from class CSubject. We borrow an abstraction technique from [10] (but omit some of the associated details). Abstract fields and invariants can be introduced by the keyword **abstract**. Such fields and invariants can be implemented differently by different subclasses. The specification of a method is inherited by an overriding method. The need to include inherited flanking invariant J as a conjunct of I is a technical detail that is due to the admissibility obligation (see section 5).

4.2 Generalized coop-sets

The main advantage of the generalization is that it makes the **coop** construct 'sufficiently expressive'. Assume that field f is specified in class C. Assume that class D defines an invariant with name I. When name I is *not* hidden from C, invariant I of an arbitrary object of class C is cooperated with by field f when its coop-set includes $(D, I, true)$. Additional conditions on this arbitrary object can be specified by a predicate other than $true$. For instance, the specification of field d of class CSubject can be changed to **int** d **coop** (Observer, I, $\exists i \bullet \mathbf{inc} = \mathbf{this}.on.next^i.obs$). This specifies that only Observers in the list on maintained by CSubject **this** can be invalidated. When the name I of the invariant *is* hidden from class C, no invariant I of any object of class D can be cooperated with by field f. However, it should not be allowed that an assignment in a method M to a field f invalidates such an invariant I as this would make most designs infeasible. Therefore, it is not a restriction that such an invariant cannot be cooperated with.

5 Formalization of the Proof Technique

The proof technique introduced in this section can complement any proof system that proves correctness of statement annotations (that has some standard properties for logical variables). Given this complemented proof system, any execution of a program that is proven correct meets the premises of Theorem 3.1. Hence, the conclusion of the theorem allows to assume the validity of (most) invariants when a method is called or terminates.

5.1 Terminology

We first introduce basic terminology. A program consists of a set of classes. C and D identify classes as before. A class defines a set of fields, methods and invariants. All are inherited when a class is extended by a subclass. f identifies a field. $coop(f, C)$ yields the coop-set of field f defined in class C (section 4.2). For simplicity, defining a subclass field with the same name as a superclass field (field shadowing) is disallowed (removing this restriction results in a number of additional typecasts). M_C identifies method M defined in class C. $inc(M_C)$ yields the inc-set of method M_C (see section 3). When M_C overrides M_D, $inc(M_C)$ must be a superset of $inc(M_D)$. α identifies an object, i.e. the instantiation of a class (think of α as an address). A *location* $\alpha.f$ stores the value of object α's field f. In Java-like languages, objects and their contents are accessed only by *references*. A reference r consists of a *scope variable* and zero or more field accesses of the form $.f$. A scope variable s_C is a *local variable* l_C, a *method parameter* p_C or a *logical variable* X_C (for convenience, the static type C of a variable is made explicit). Every method of a class C implicitly defines a parameter this_C. $(C)r$ denotes the typecast of reference r to class C. P identifies a predicate. When $\bar{r}$ and $\bar{r}'$ are vectors of references, $P[\bar{r}/\bar{r}']$ denotes the simultaneous, capture-avoiding substitution of $\bar{r}'$ by $\bar{r}$ in predicate P. An *invariant* is a tuple (I, P), with I a name that identifies the invariant and P a predicate in which only **this** occurs as scope variable. The restriction on P guarantees that an invariant can only be invalidated by an assignment statement (in particular, not by object creation). Reference $r.f$ is called a *supplier reference* of an invariant (I, P) when $r.f$ either occurs in P or is a subreference of a reference that occurs in P. For simplicity and due to lack of space, we omit our treatment of supplier references of the shape $r.f^i$ (section 3.2) and only allow references of the shape $r.f$ in P. The names of the invariants of a class C are distinct. For convenience, these names are also assumed distinct from the names of invariants defined in superclasses of C. An *object invariant* $I(\alpha)$ denotes invariant I of object α. A reference invariant $I(r)$, where I is the name of an invariant defined in (a supertype of) the static type of reference r, may occur in a predicate (section 3.1).

5.2 Representing Control Flow

The presentation of the proof obligations is orthogonal to the rest of the proof system, i.e., it is assumed that every method body is fully and correctly annotated. When, during method execution, control is at a certain program location, the corre-

sponding predicate holds in that state. The proof obligations force the annotation to be of a certain shape. To formulate the proof obligations, a high-level abstraction of the grammar of (fully) annotated statements suffices. We use the following grammar:

$$\begin{aligned} S &::= \{P\} \mid \{P\}Stat;\ S \\ Stat &::= Basic \mid \mathbf{if}(S, S') \mid \mathbf{while}(S) \\ Basic &::= \mathbf{assign}(r) \mid \mathbf{mc}(r, M_C) \mid \mathbf{new}(l_C, D) \mid \mathbf{dtc} \end{aligned}$$

The statement $\mathbf{assign}(r)$ is an assignment to reference r, where the right-hand side is a reference or a primitive value. $\mathbf{mc}(r, M_C)$ is a method call to method M_C with reference r as receiver (the call might be dynamically dispatched to a subclass-method M_D). For simplicity, method parameters (other than this) are not allowed. Removal of this restriction is straightforward. The use of object creation statement $\mathbf{new}(l_C, D)$ and constructor-only statement **dtc** is explained in section 5.3.

In different executions of a method, control can flow through the method in different ways. Consider fully annotated method m (using square brackets for annotation):

m(){[P_0] this.f := 1; [P_1] **if** (a == b) { [P_2] this.g.m() [P_3]}; [P_4] this.h := 0; [P_5]}

Suppose this.f := 1 can invalidate an invariant (field f has a non-empty coop-set). Then this invariant must either be implied by P_1 or P_2 (i.e., re-established before the call) or be in the inc-set of $this.g.m()$. It must also be implied by either P_1, P_4 or P_5 (i.e., re-established before the end of the method). We associate a graph with a method to express which methods may be called between two statements. Control flow through the method is represented by a path in the graph.

$body(M)$ yields M's body, the annotated statement S. In each state of a method execution, control is at a specific program location in this method. Each program location identifies exactly one annotated statement S, which is represented as a node in a graph. A node n is a tuple $(P, eStat)$, where $eStat$ identifies an element of the set $\{Stat, \mathbf{end}\}$. A program location that identifies an annotated statement $\{P\}$ is represented by a node $(P, \mathbf{end})$. A program location that identifies an annotated statement $\{P\}Stat; S$ is represented by a node $(P, Stat)$. Two functions on a node are defined.

$$pre((P, eStat)) \stackrel{def}{=} P \text{ and } stat((P, eStat)) \stackrel{def}{=} eStat$$

An *edge* e is a tuple (n, n'). When there is an edge (n, n') in the graph, the program location n' can be reached from program location n in a single execution step. When the program counter identifies n in a particular execution state, in the next execution state it identifies a node n' such that (n, n') is an edge in the graph. A graph G is a tuple of a set of nodes N and a set of edges E. A union on graphs is defined: $(N, E) \cup (N', E') \stackrel{def}{=} (N \cup N', E \cup E')$. The graph G of an annotated statement contains exactly one node without incoming, and one node without outgoing edge. $start(G)$ and $end(G)$ yield these nodes. $graph(S)$, defined

$$
\begin{array}{ll}
graph(\{P\}) & \stackrel{def}{=} \text{Let } N = \{(P, \mathbf{end})\} \text{ in } (N, \{\}) \\
graph(\{P\}Basic; S) & \stackrel{def}{=} \text{Let } G = graph(S) \text{ and } n = (P, Basic) \text{ in } G \cup \{\{n\}, \{(n, start(G))\}\} \\
graph(\{P\}\mathbf{if}(S_1, S_2); S_0) & \stackrel{def}{=} \text{Let } G_0 = graph(S_0) \text{ and } G_1 = graph(S_1) \text{ and } G_2 = graph(S_2) \\
 & \quad \text{and } n_0 = (P, \mathbf{if}(S_1, S_2)) \text{ and } n_1 = start(G_0) \text{ and} \\
 & \quad E = \{(n_0, start(G_1)), (n_0, start(G_2)), (end(G_1), n_1), (end(G_2), n_1)\} \\
 & \quad \text{in } G_0 \cup G_1 \cup G_2 \cup \{\{n_0\}, E\} \\
graph(\{P\}\mathbf{while}(S_1); S_0) & \stackrel{def}{=} \text{Let } G_0 = graph(S_0) \text{ and } G_1 = graph(S_1) \text{ and } n_0 = (P, \mathbf{while}(S_1)) \\
 & \quad \text{and } n_1 = start(G_0) \text{ and } n_2 = start(G_1) \text{ and } n_3 = end(G_1) \\
 & \quad \text{in } G_0 \cup G_1 \cup \{\{n_0\}, \{(n_0, n_1), (n_0, n_2), (n_3, n_1), (n_3, n_2)\}\}
\end{array}
$$

Fig. 7. graph construction algorithm

in Figure 7, yields the graph of annotated statement S.

$$
\begin{aligned}
start((N, E)) &= n \text{ iff } n \in N \text{ and } \forall n' \bullet (n', n) \notin E \\
end((N, E)) &= n \text{ iff } n \in N \text{ and } \forall n' \bullet (n, n') \notin E
\end{aligned}
$$

Seq identifies a sequence. $|Seq|$ yields the length of sequence Seq. $Seq[i]$ yields the i'th element of Seq. $Seq[i, j]$ yields the subsequence of Seq of elements i up to and including j. $Seq[i, j)$ yields the subsequence of Seq of elements i up to but not including j. $Seq[i..)$ yields the postfix of Seq that starts at element i. A sequence of nodes $nSeq$ is *a path in* graph (N, E) when the nodes in the sequence are (pair-wise) adjacent, i.e., when $nSeq[0] \in N$ and $\forall i \bullet (0 < i < |nSeq| \Rightarrow (nSeq[i-1], nSeq[i]) \in E)$. When $nSeq$ is a path in graph G, it is *a path from* node n when $nSeq[0] = n$ and it is *a path to* node n when $nSeq[\,|nSeq| - 1] = n$. $nSeq$ is *cycle-free* when all its elements are distinct, i.e, $\forall i, j \bullet (0 \leq i < j < |nSeq| \Rightarrow nSeq[i] \neq nSeq[j])$.

An *execution sequence* Θ is a sequence of *states*. A state is a tuple (τ, n, Θ'), where τ consists of a heap and a stack. $nodes(\Theta)$ is the sequence of nodes $nSeq$ such that when $\Theta[i] = (\tau, n, \Theta')$, $nSeq[i] = n$. Let $graph(body(M_C)) = G$. Then Θ represents an execution of M_C when 1) $nodes(\Theta)$ is a path from $start(G)$ in G that is either infinite or is a path to $end(G)$ and 2) when $\Theta[i] = (\tau, n, \Theta')$, Θ' is the empty sequence unless $stat(n)$ is of the form $\mathbf{mc}(r, M'_D)$ (then Θ' represents an execution of method M'_D). So, when $graph(body(M_C)) = G$, any possible way control can flow in a (terminating) execution of method M_C is represented by a path from $start(G)$ to $end(G)$ in G.

Recall that s_C is a scope variable of static type C. A node $(P, eStat)$

establishes $I(s_C)$ iff $P \Rightarrow I(s_C)$

respects $I(s_C)$ iff if $eStat$ is of the form $\mathbf{mc}(r, M_D)$, then

there is a P' such that $(C, I, P') \in inc(M_D)$ and $P \Rightarrow P'[(s_C, (D)r)/(\mathbf{inc}, \mathsf{this}_D)]$

A sequence of nodes $nSeq$

respects $I(s_C)$ iff $\forall i \in nSeq \bullet nSeq[i]$ respects $I(s_C))$

establishes $I(s_C)$ iff $\exists i \in nSeq \bullet (\ nSeq[i]$ establishes $I(s_C)$ and

$nSeq[o..i)$ respects $I(s_C)\ )$

is safe for $I(s_C)$ iff $nSeq$ respects or establishes $I(s_C)$

In an execution represented by a path that respects $I(s_C)$, no method is called that relies on the object invariant represented by $I(s_C)$. In an execution represented by a path that establishes $I(s_C)$, there is an execution state in which the object invariant represented by $I(s_C)$ holds and no method that relies on that object invariant is called before that state.

5.3 Constructors

The semantics of constructors is treacherous due to dynamic binding. The first (possibly implicit) statement in a constructor of class C is a call to the constructor of C's superclass. When the superclass constructor calls a method, and it is dynamically dispatched to an overriding method in a subclass, this method might rely on an invariant yet to be established. We do not consider this good OO design, but it is possible in Java. Rather than (slightly) changing Theorem 3.1 and introducing restrictions on programs to prevent such implementations, we use a different (more logical) semantics for constructors [9]. The body of a constructor of class C is of the shape $\{P_0\}\mathbf{mc}(\mathsf{this}, M_D); \{P_1\}\mathbf{dtc}; S$, where $\mathbf{mc}(\mathsf{this}, M_D)$ is a call to the constructor of C's superclass. $\mathbf{new}(l_C, D)$ creates an object of class Object and calls the constructor of class D to initialize the object. Statement **dtc** (dynamic type change) in a constructor of class C changes the dynamic type of the object that is initialized to class C. No invariants of the object that is initialized are invalid when the superclass constructor is called (class Object does not define any). This semantics of constructors is similar to that of C++.

5.4 Proof Obligations

Finally, the proof technique for invariants can be presented. The admissibility obligation below guarantees that, when a location $\alpha.f$ is assigned to, any object invariant *not* cooperated with by f is *not* invalidated by the assignment. A proof relies on the definition of coop in section 4.1 and a context switch.

Proof Obligation 1 (Admissibility) *For every invariant* (I, P) *defined in a class* C*, for every supplier reference* $r.f$ *of* (I, P)*,*

if $r.f$ *refers to field* f *specified in class* D,

then there is a $(C, I, P') \in coop(f, D)$ *such that* $P \Rightarrow P'[(\mathbf{this}, (D)r)/(\mathbf{dep}, \mathbf{this})]$

In a setting where information hiding for invariants (or more generally, modular development) is not required, this obligation and the **coop** construct are not needed. To still apply the proof technique, the default coop-set can be changed to include $(C, I, true)$ for every class C in the program, for every invariant I defined by class C. This trivializes the admissibility obligation. For simplicity, we omit a weaker version of the obligation. For instance, one can capitalize on the fact that, in a state where P doesn't hold, an invariant $(P \vee P')$ cannot be invalidated by an assignment to a field that only occurs in P.

Before the remaining proof obligation are presented, a theorem is formulated that is essential for both the soundness of the approach and the intuition behind it.

Theorem 5.1

When every cycle-free path from n' *in* G *is safe for* $I(X_C)$, *and*

every cycle-free path from n' *to* $end(G)$ *in* G *establishes* $I(X_C)$

Then every path from n' *in* G *is safe for* $I(X_C)$, *and*

every path from n' *to* $end(G)$ *in* G *establishes* $I(X_C)$

Proof. Straightforward (by induction on the number of cycles in an arbitrary path) □

As this theorem shows, only cycle-free paths have to be considered by the proof obligations.In the remainder of this section, three more proof obligations are introduced.

Proof Obligation 2 (Constructor) *For every constructor* M *of a class* C, *for every invariant* (I, P) *defined in* C,

if $graph(body(M)) = G = (N, E)$ *and* $\{(start(G), n), (n, n')\} \subseteq E$,

then every cycle-free path from n' *in* G *is safe for* $I(\mathsf{this}_C)$, *and*

every cycle-free path from n' *to* $end(G)$ *in* G *establishes* $I(\mathsf{this}_C)$

Here, node n' is the program location directly after the **dtc** statement (section 5.3). This proof obligation establishes that, no matter how control flows through a constructor, the invariant of the newly constructed object is established, and no method that relies on it is called before this is done, which is needed for the third premise of Theorem 3.1. One way to establish that this proof obligation is met is to use a breadth-first algorithm that searches for a node that establishes the invariant (and stops after a full cycle is traversed). For instance, $apv(n, G, I(s_C))$ implies that every cycle-free path from n in G is safe for $I(s_C)$ and that every cycle-free path from n to $end(G)$ in G establishes $I(s_C)$. A proof by induction on the length of cycle-free paths is straightforward.

$apv(n,(N,E),I(s_C)) \stackrel{def}{=} n \notin N$ or n establishes $I(s_C)$ or ($n \neq end((N,E))$ and n respects $I(s_C)$ and (for every edge $(n,n') \in E, apv(n',(N-\{n\},E),I(s_C))$ holds))

The two other proof obligations use logical variables that keep track of which invariants might be invalid.

Proof Obligation 3 (Inc) *For every method M of a class C, for every $(C,I,P) \in inc(M)$, if $graph(body(M)) = G$, then there exist a predicate P' and a logical variable X_C such that*

X_C does not occur free in P', and

$pre(in(G)) \Leftrightarrow (P' \wedge (X_C = null \vee P[X_C/\mathbf{inc}]))$, and

every cycle-free path from $in(G)$ in G is safe for $I(X_C)$

Let the coop-set of method M include (C,I,P). Then, for an arbitrary object X_C such that $P[X_C/\mathbf{inc}]$ holds in the prestate of an execution of M the following property is established. No matter how control flows through method M, no method that relies on $I(X_C)$ is called unless $I(X_C)$ has been established to hold. Note that $X_C = null \vee P[X_C/\mathbf{inc}]$ can always be assumed to hold (it says, either there is an arbitrary object X_C such that $P[X_C/\mathbf{inc}]$ holds, or there is not). A small example: Let $inc(M)$ include $(C,I,false)$. Then, $(X_C = null \vee P[X_C/\mathbf{inc}]) \Rightarrow X_C = null$. As $I(null)$ is trivially true, $in(G)$ establishes $I(X_C)$ and the proof obligation is met (of course, $(C,I,false)$ is not a sensible inclusion in an inc-set).

Proof Obligation 4 (Cooperation) *For every method M of a class C,*

if $graph(body(M)) = G = (N,E)$, and $(n,n') \in E$, and $stat(n) = \mathbf{assign}(r.f)$),

and $r.f$ refers to field f of class D, and $(C,I,P) \in coop(f,D)$,

then there exist a predicate P' and a logical variable X_C such that

X_C does not occur free in P', and

$pre(n) \Leftrightarrow (P' \wedge (X_C = null \vee P[(X_C,(D)r)/(\mathbf{dep},\mathbf{this})]))$, and

every cycle-free path from n' in G is safe for $I(X_C)$, and

every cycle-free path from n' to $end(G)$ in G establishes $I(X_C)$

This obligation combines the techniques of the previous two to deal with assignment statements. Note that, when n is an assignment node, there is exactly one outgoing edge (n,n'). An invariant that does not hold in the precondition of a method is trivially preserved. For simplicity, this is not capitalized on by the proof obligation.

5.5 Soundness

Lack of space prevents presentation of a full soundness proof (i.e., the premises of Theorem 3.1 are met given the proof obligations). We sketch a proof that when an assignment in a method execution invalidates an invariant $I(\alpha_1)$, it is

re-established before the method terminates, and that any method called before $I(\alpha_1)$ is re-established specifies that it does not rely on $I(\alpha_1)$. In a similar way, one can prove that, when an invariant is invalid when a method M_C is called, any method that is called by M_C while the invariant is invalid specifies that it does not rely on it, and that constructors establish their invariants.

A *logical environment* ω maps logical variables to values. $\tau, \omega \models P$ iff P holds given the mappings in τ and ω. $\omega[X_C \rightarrow \alpha]$ is the state like ω, but with X_C mapped to α. Let Θ_0 represent an arbitrary method execution. Let $\Theta_0[i] = (\tau_0, n, [])$. Let $n = \mathbf{assign}(r_0.f)$, where f is defined in class D. Let τ_0 map r_0 to α_0. Assume invariant $I(\alpha_1)$ is invalidated by the assignment. Assume this invariant is defined in class C as (I, P_0). Then $I(\alpha_1)$ holds in the prestate, i.e., $\forall\omega \bullet \tau_0, \omega[X_C \rightarrow \alpha_1] \models P_0[X_C/\mathbf{this}]$ (as **this** is the only scope variable in P_0). As only $\alpha_0.f$ is changed by the assignment, P_0 must have a supplier reference $r_1.f$, with f defined in C, that refers to $\alpha_0.f$. Then, due to the admissibility obligation, there is a $(C, I, P_1) \in coop(f, D)$ such that $P_0 \Rightarrow P_1[(\mathbf{this}, (D)r_1)/(\mathbf{dep}, \mathbf{this})]$. Then (using two context switches) $\forall\omega \bullet \tau_0, \omega[X_C \rightarrow \alpha_1] \models P_1[(X_C, (D)r_0)/(\mathbf{dep}, \mathbf{this})]$ (as **dep** and **this** are the only scope variables in P_1). Let $nodes(\Theta_0) = nSeq$. Assume the method execution terminates. From Theorem 5.1 and the cooperation obligation it follows that there is a $j > i$ such that $nSeq[j]$ establishes $I(X_C)$ and $nSeq(i..j]$ is safe for $I(X_C)$. The *main theorem* relied on for soundness is the following: Let $\Theta[i] = (\tau, n, \Theta')$ and $j > i$ and $\Theta[j] = (\tau', n', \Theta'')$. Then $\forall\omega \bullet (\tau, \omega \models pre(n)$ implies $\tau', \omega \models pre(n'))$. A proof relies on the correctness of the annotation, the soundness of the proof system and some (fairly standard) assumptions about logical variables. From this theorem and the above it follows that, when $\Theta_0[j] = (\tau_1, n_1, \Theta_1)$, then $\forall\omega \bullet \tau_1, \omega[X_C \rightarrow \alpha_1] \models I(X_C)$, which means $I(\alpha_1)$ is valid in $\Theta[j]$. To prove that methods called do not rely on $I(\alpha_1)$, assume that there is a k, $i < k < j$, such that $\Theta_0[k] = (\tau_2, n_2, \Theta_2)$ and $stat(n_2) = mc(r_2, M_D)$ (for some class D) and r_2 refers to α_3 in τ_2. Then, due to the cooperation obligation and Theorem 5.1, n_2 respects $I(X_C)$ and therefore, there is a P_2 such that $(C, I, P_2) \in inc(M_D)$ and $pre(n_2) \Rightarrow P_2[(X_C, (D)r_2/(\mathbf{inc}, \mathsf{this}_D))]$. Due to the main theorem above, $\forall\omega \bullet \tau_2, \omega[X_C \rightarrow \alpha_1] \models pre(n_2)$. Therefore, $\forall\omega \bullet \tau_2, \omega[X_C \rightarrow \alpha_1] \models P_2[(X_C, (D)r_2/(\mathbf{inc}, \mathbf{this}))]$. Thus, when $\Theta_2 = (\tau_3, n_3, \Theta_2)$ (as **inc** and this are the only scope variables in P_2), $\forall\omega \bullet \tau_3, \omega[\mathbf{inc} \rightarrow \alpha_1] \models P_2$. As any method overriding M_D also includes (C, I, P_2) in its inc-set, this proves that the method execution specifies that it does not rely on $I(\alpha_1)$. Finally, when method execution Θ_0 does not terminate, it follows that $nSeq[i..)$ is infinite and safe for $I(X_C)$, and the proof is similar to the one above.

This concludes our formal treatment. Note that proof rules that allow invariants to be assumed at certain points in the proof are omitted but these are straightforward.

6 Related and Future work

Ownership-based approaches and our approach are complementary (an object owns another when it has some form of control over access to the other's data), and

combining them should be fairly straightforward. Several ownership mechanisms have been proposed (e.g., [15,6,10]). Given modular development, a complementary approach is needed as invariants of which the name is hidden from a class C 1) cannot be in the inc-set of a method of C and 2) cannot always be expected to be preserved by methods of C when the structure is hierarchical. Our *coop* construct is similar to the explicit dependencies in [10], which generalizes earlier work in [5]. However, these do not allow a specification as in Figure 6. A liberal, but semantical admissibility obligation on invariants is used in [10]. In the Boogie ([6]) and the friendship ([2]) approaches (both extending [1]), as well as in [12], flexible abstraction mechanisms are provided that allow a method to specify that it relies on a hidden property, and that allow a user to track the validity of such a hidden property. While very useful for information hiding, these do not prevent the propagation of (abstractions of) properties. We expect the friendship approach can achieve specifications of similar strength, without propagation, when one adds either 1) an inc-like construct to specify which inv-bits do not hold in a precondition and the program invariant that all other inv-bits hold or 2) a default precondition that all inv-bits hold and the possibility to override this default. Compared to cooperation-based invariants, this gives a less intuitive semantics for invariants but a more intuitive proof technique. However, it has the additional overhead of both pack/unpack and attach/detach. The work in [14] shows uses of invariants that quantify over (unreachable) objects, and how they can be allowed. The premises of theorem 3.1 do not disallow such invariants. An extension that allows such invariants is considered future work.

7 Conclusion

Data induction allows a method to rely on an invariant without it being specified in pre- and postconditions. We have introduced an approach that allows this for invariants over object structures. The **inc** construct specifies that a method does not rely on certain invariants. We argue that this is essential for the specification of many natural, non-hierarchical designs. The **coop** construct specifies which invariants can be invalidated by an assignment. This allows the verification of invariants even when their definition is hidden. In particular, this makes the approach suitable to modular development. We introduce proof obligations that guarantee data induction is allowed.

References

[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, June 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[2] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Seventh International Conference on Mathematics of Program Construction (MPC)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag, 2004.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[5] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995.

[6] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.

[7] 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.

[8] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition.* Prentice-Hall, Englewood Cliffs, New Jersey, 1997.

[9] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based invariants for oo languages. In *International Workshop on Formal Aspects of Component Software (FACS'05)*, 2005.

[10] Peter Müller. *Modular Specification and Verification of Object Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[11] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[12] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of 32nd POPL*, volume 40 of *ACM SIGPLAN Notices*, pages 247–258, January 2005.

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[14] Cees Pierik, Dave Clarke, and Frank S. de Boer. Controlling object allocation using creation guards. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2005.

[15] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic ownership. In *Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[16] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Third Annual Symposium on Logic in Computer Science (LICS)*, 2002.