Available online at www.sciencedirect.com
SCIENCE DIRECT
Science of Computer Programming 62 (2006) 253–286
Science of Computer Programming
www.elsevier.com/locate/scico

Modular invariants for layered object structures

Peter Müller[a,*], Arnd Poetzsch-Heffter[b], Gary T. Leavens[c]

[a] *ETH Zurich, Switzerland*
[b] *Technische Universität Kaiserslautern, Germany*
[c] *Iowa State University, Ames, IA, USA*

Received 9 March 2005; received in revised form 10 January 2006; accepted 29 March 2006
Available online 19 May 2006

Abstract

Classical specification and verification techniques support invariants for individual objects whose fields are primitive values, but do not allow sound modular reasoning about invariants involving more complex object structures. Such non-trivial object structures are common, and occur in lists, hash tables, and whenever systems are built in layers. A sound and modular verification technique for layered object structures has to deal with the well-known problem of representation exposure and the problem that invariants of higher layers are potentially violated by methods in lower layers; such methods cannot be modularly shown to preserve these invariants.

We generalize classical techniques to cover layered object structures using a refined semantics for invariants based on an ownership model for alias control. This semantics enables sound and modular reasoning. We further extend this ownership technique to even more expressive invariants that gain their modularity by imposing certain visibility requirements.

Keywords: Invariant; Object-oriented programming; Modular verification; Ownership; Visibility; JML

1. Introduction

Invariants are predicates that specify what states of an object are consistent [1]. Typically, they restrict the range of values that an instance variable may take and, more importantly, express that the values of two or more variables have to be related in a certain way. For example, the invariant of the `List` class, near the top of Fig. 1, states several such properties, including that the `array` field is always non-null and that the unused part of the array holds zeros. Thus, when calling `add`, for example, the expression `array.length` cannot cause a null pointer exception. Even if not formally specified, invariants help in writing, understanding, maintaining, and reasoning about programs [2].

In this paper we discuss invariants in technical terms, which allows us to prove the validity of the techniques. A detailed formalization of the language, its semantics, and all proofs is presented in [3]. In this paper, we omit the formalization of the language and type system and concentrate instead on the key points that enable modular verification of invariants. In addition we also provide more intuition and background. In particular, the programming

* Corresponding address: ETH Zurich, ETH Zentrum, RZ F2, 8092 Zurich, Switzerland. Tel.: +41 44 63 22868; fax: +41 44 63 21435.
E-mail addresses: peter.mueller@inf.ethz.ch (P. Müller), poetzsch@informatik.uni-kl.de (A. Poetzsch-Heffter), leavens@cs.iastate.edu (G.T. Leavens).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.03.001

guidelines we propose in the conclusions (Fig. 17) are relevant for programmers, even if they do not use formal techniques to reason about their programs.

1.1. Problems and their importance

The overall problem we address in this paper is how to generalize classical reasoning techniques for invariants to deal with layered object structures while retaining modularity. More specifically, we address the problems of representation exposure and when invariants for layered object structures should hold.

In the last twenty years, several techniques for reasoning about class invariants have been developed (see for example [4–8]). Most of these techniques are based on a simplification that tightly restricts their applicability in practice. They assume, often implicitly, that the only invariants a method of a class C potentially breaks are the invariants of its receiver object or of other C objects. These techniques are usually sound for invariants that *depend* only on a single object whose fields are primitive values, such as points with integer coordinates. Although they are different in detail and often neglect to explicitly state requirements for reasoning, we refer to them as the classical verification techniques or simply as the *classical technique*. For practical use, the classical technique is too restricted, mainly for two reasons:

(1) Many interesting invariants depend on several objects.
(2) A method of class C is not constrained to modify only C objects. It can modify all its parameter objects and all objects that are reachable from the parameters, for instance, by assigning to non-private fields or to array elements.

As a simple example illustrating some of the resulting problems, we consider class `List` of Fig. 1. Its invariant depends on locations of the underlying array object. Thus, any method in any class that gets hold of the array reference can access the array and break `List`'s invariant. To avoid such situations, we have to prevent the array reference from being exposed to other objects. This gives a first idea why modular reasoning about invariants is linked to the problem known as *representation exposure* [4,9,10].

Furthermore, the classical technique is not sound for invariants of layers implemented on top of `List`. For example, the invariant of class `BagWithMax` in Fig. 2, which says that no element of the list is larger than a given upper bound, is generally not preserved by `List`'s `add` method. The requirement that `BagWithMax`'s invariant should hold in the poststate of `add` cannot be handled modularly, since modular verification of a class implies not considering its clients during its verification.

These problems are important because invariants that depend on the state of objects of an underlying layer are common. They occur in three situations. The first is when invariants of the upper layer relate locations in the upper layer and the object states in the underlying layers, as illustrated by `BagWithMax`. The second is when an upper layer restricts the object states of the underlying layers. For instance, a set built on top of a list might have an invariant that excludes duplicates in the list. The third is when the invariant of an upper layer relates the states of different objects of an underlying layer. This is often the case in aggregate objects. For example, consider `Family` objects that aggregate different `Person` objects. `Family`'s invariant could require that all `Persons` in a `Family` have the same street address.

1.2. Approach

Our approach is to use an ownership model that imposes a hierarchical context structure on the heap. This model allows us to address the soundness problem due to representation exposure by controlling the references into a context. To address the modularity problem due to layering, we use the context hierarchy to define a novel semantics of invariants. This semantics frees methods of lower layers from the obligation to preserve the invariants of higher layers.

In our example, a `BagWithMax` object b owns the `List` object, say l, referenced by b.`theList`. This ownership relation prevents other objects from modifying l, which prevents them from breaking b's invariant and thus avoids the negative effects of possible representation exposure.

By our semantics of invariants, this ownership relation also allows methods executed on l to violate the invariant of b. However, this invariant can be re-established by the method of b that called the method of l. This shows the modularity of our approach: the proof obligations for b's invariant fall on the methods of b, not on the methods of l. In other words, methods of lower layers do not have to know about properties of higher layers, which is one of the key principles of layered designs [11, Sec. 6.3.4].

```
class List {
  private /*@ spec_public rep @*/ int[] array;
  private /*@ spec_public @*/ int n;

  /*@ public invariant array != null
    @      && 0 <= n && n <= array.length
    @      && (\forall int i; 0<=i && i<n; array[i]>=0)
    @      && (\forall int i; n<=i && i<array.length; array[i]==0);
    @*/

  /*@ requires k >= 0 && n < Integer.MAX_VALUE;
    @ assignable array, array[n], n;
    @ ensures n==\old(n+1)
    @     && array[\old(n)]==k
    @     && (\forall int i; 0<=i && i<\old(n);
    @                    array[i]==\old(array[i]));
    @*/
  public void add(int k) {
    if (n==array.length) { resize(); }
    array[n] = k; // temporary invariant violation
    n++;
  }

  //@ assignable array, n;
  public void resize()                { /* ... */ }
  public List() { array = new /*@ rep @*/ int[10]; }
  public void addElems(int[] elems)   { /* ... */ }
  public /*@ pure @*/ int  size()     { return n; }

  // other methods omitted.
}
```

Fig. 1. Implementation of an array-based list. Annotation comments start with an at-sign (@), and at-signs at the beginning of lines are ignored. The array object is part of the encapsulated internal representation of the list, indicated by the `rep` annotation, which will be explained in Section 5.2. The `spec_public` annotation allows fields with any access modifier to be mentioned in public specifications. The `pure` annotation says that a method cannot have side effects, and allows it to be called in assertions.

```
class BagWithMax {
  private /*@ spec_public rep @*/ List theList;
  private /*@ spec_public @*/ int maxElem;

  /*@ public invariant theList != null
    @        &&   (\forall int i; 0<=i && i<theList.n;
    @                theList.array[i] <= maxElem);
    @*/

  //@ requires k>=0;
  public void insert(int k) {
    theList.add(k);
    if (k > maxElem) { maxElem = k; }
  }

  // other methods and constructors omitted.
}
```

Fig. 2. Class `BagWithMax` builds an abstraction layer on top of `List`. `BagWithMax`'s invariant depends on the state of the `List` object and its array.

Based on this approach, we present two related verification techniques. Our ownership technique (Section 6) uses the ownership model to describe what fields are allowed to be mentioned in an invariant, to say when invariants have to hold, and to devise the proof obligations that are necessary for soundness. The visibility technique (Section 9) extends the ownership technique. It uses the structure of the program to trade additional proof obligations for more expressive invariants. In particular, these invariants can express properties of structures that do not strictly follow the ownership model.

Before giving the details of these techniques, we start by presenting some necessary background. In the rest of this section we give a brief overview of the most closely related work, in order to highlight this paper's contributions. We also describe our assumptions about the programming and specification language. Section 2 describes background on the invariant problem and the way we characterize its semantics. Section 3 describes the classical technique for reasoning about invariants. Its problems and limitations are discussed in Section 4. Then Section 5 presents background on the ownership model. Section 6 uses the ownership model to give a first solution technique, namely the ownership technique. Section 7 extends this solution with transitive readonly references for more flexibility. Section 8 describes limitations of this ownership technique, which motivates the more flexible visibility technique, which is described in Section 9. Section 10 discusses the tradeoffs among these solutions. The paper ends with sections on future work, related work, and conclusions.

1.3. Contributions to the state of the art

A verification technique for invariants defines the execution states of a program in which invariants must hold. We cannot require that invariants hold in all execution states because invariants that depend on two or more variables potentially do not hold while these variables are updated sequentially. For example in Fig. 1, method `add` might temporarily violate the invariant if the added integer is strictly positive. Then, in the state before incrementing `n`, the last conjunct of the invariant does not hold. Of course, the invariant is re-established before the method terminates.

A direct approach to solving this *temporary violation problem* of invariants is given by the Boogie technique [12, 13]. The essence of this technique is an explicit treatment of when invariants must hold in specifications. In the Boogie technique, an object is either in a valid or a mutable state. Only when in a valid state, an object has to satisfy its invariant, and only objects in a mutable state can be modified. The additional state information is represented by a specification-only field, which is modified by two special statements, `pack` and `unpack`. To handle invariants for object structures, objects are organized in an ownership hierarchy. For instance, a `List` object would own its underlying array object, and a `BagWithMax` object would own its `List` object. The Boogie technique enforces that all (transitive) owner objects of a mutable object are also mutable. Therefore, when an object is modified, the invariants of its owner objects need not hold. In our example, a modification of the array object would require that the array, the owning `List` object, and the `BagWithMax` object owning the list, be mutable. Therefore, the modification would be allowed to break the invariants of these objects. The invariants would be checked when the objects are packed, that is, turned from mutable to valid.

Whereas the explicit representation of object validity and the `pack` and `unpack` statements allow the Boogie technique to handle invariants for layered object structures, they also increase the specification overhead significantly: it is non-trivial to figure out where to use the new statements, and method specifications become much more complex, due to the need to describe what objects are valid, that is, what invariants are expected to hold.

Although developed earlier than the Boogie technique, our ownership technique can be regarded as a combination of the classical and Boogie techniques. Our technique's approach, adopted from Müller's thesis [3], is to define an invariant semantics based on an ownership model for alias control. Like the classical technique, it statically defines the execution states in which an invariant most hold, avoiding the overhead of the Boogie technique's explicit treatment of validity. In our ownership technique, the ownership model is enforced by a type system that also prevents representation exposure problems.

Müller's thesis also gives an indirect treatment of invariants that solves this problem. That treatment is indirect because it is based on a translation into specification-only fields, and relies for soundness on a technique for modular verification of modifies clauses. In this paper, we give a simpler, direct formal treatment that uses no such translation and is independent of modifies clauses. This direct treatment results in several advantages, including less specification overhead and an overall simplification of the soundness proof.

Another contribution of this paper is that we clearly explain the relation between the semantics of invariants, the locations an invariant is allowed to depend on, and the encapsulation that is necessary to reason about invariants modularly. This helps explain what restrictions are necessary for classical techniques [8,4–7] to be (or to become) sound.

Such classical techniques are modular due to the simple way that they encapsulate the locations an invariant may depend on. Another contribution of our work is that we clarify the distinction between these techniques and techniques that gain modularity by imposing certain visibility requirements on such fields [14,15]. We both present the details of the encapsulation-based approaches, and discuss how to generalize such reasoning techniques to invariants that are based on visibility requirements.

1.4. Assumptions

The techniques we present depend, in their technical details, on a few assumptions about programming and specification languages. Some of these assumptions are made to avoid complications in the proofs. We discuss how to relax some of these assumptions at various points in the paper.

For the programming language, we assume a sequential Java subset including classes, inheritance, arrays, instance fields, instance methods, statements, and expressions. For simplicity, we omit interfaces, exceptions, static fields, static methods, and inner classes. Müller shows how to handle interfaces, exceptions, and static methods [3,16]. An extension to static fields is future work.

Although we focus on sequential programs in this paper, we expect that our results are also applicable to multi-threaded programs. Such programs can be verified by first proving atomicity by other means, which then allows one to apply verification techniques for sequential programs [17–21].

For the specification language, we use a tiny variant of JML [22,23], including requires and ensures clauses (pre- and postconditions), assignable (i.e., modifies) clauses, and invariants. As shown in Fig. 1, the *invariant of a class* is declared by an invariant clause of the form `invariant` I`;` where I is essentially a side-effect free boolean expression of the programming language that may contain reads from fields and array elements, the usual unary and binary operators, and bounded quantification expressions.

For simplicity, we prohibit method invocations and object creation in specifications. JML only allows side-effect free calls in specifications, which can be formally modeled by applications of mathematical functions. Müller's thesis [3] shows that such applications in invariants can be handled by explicitly declaring which variables an invariant depends on. Such dependency declarations are used to determine all invariants that might be violated by a field update. However, explicit dependency declarations introduce significant specification overhead, which is a distraction in this paper.

Furthermore, to keep the presentation more readable, we omit an explicit treatment of constructors in definitions and theorems. For purposes of this paper, constructors are analogous to methods. The only exception is that a (newly created) receiver object need not satisfy its invariant in the constructor's prestate and, consequently, we cannot assume the invariant when the constructor is executed.

As in JML, an object X must satisfy the invariant of its superclasses as well as the invariant declared in its own class. The conjunction of these invariants is called the *object invariant* of X, or simply the *invariant* of X.

2. Semantic background

A technique for specifying and reasoning about invariants must address:

(1) Encapsulation: What parts of a program can assign to the variables used in an invariant?
(2) Admissible invariants: What variables may an invariant depend on?
(3) Invariant semantics: When do invariants have to hold?
(4) Modular proof techniques: How can one show that objects satisfy their invariants, without examining the entire program?

The answers to these questions closely depend on each other. However, their interconnections have not been previously described.

Answering these questions requires some general background about visible states and the concept of dependencies; these are used in all techniques presented in this paper. The particular restrictions on dependencies and how each technique answers the above questions are discussed in subsequent sections.

2.1. Visible states

As pointed out at the beginning of Section 1.3, we cannot require that invariants hold in all execution states. We follow [24,7] by using the so-called visible states to define the semantics of invariants (see for example Definition 3.3):

Definition 2.1 (*Visible States*). A program execution state is called *visible* if it is the prestate or the poststate of a method execution. The *prestate* of a method execution is the state immediately before the execution of the method body, that is, after the actual parameter values of the call have been assigned to the formal parameters of the method and control has been transferred to the method. The *poststate* of a method execution is the state immediately after the execution of the method body before control is transferred back to the caller.

Some approaches exclude the pre- and poststates of so-called "helper methods" (e.g., private methods) from the visible states [7, p. 366]. This idea can easily be added as a refinement to our approach (and is present in JML).

2.2. Dependencies

Invariants can depend on the state of various locations. Syntactically, locations are denoted by expressions either of the form $E.f$, where E is a possibly nested expression that denotes an object and f is a field, or of the form $A[i]$, where A denotes an array. To simplify notation, when X is an object, we also use the notation $X.f$ to refer to the location of the instance variable f in X. When L is a location and the object store σ is known from context, then we use the notation $L.f$ to refer to the location of the instance variable f in the object referenced from L in store σ. We use the analogous notation, $L[i]$, for arrays.

Definition 2.2 (*Dependee, Dependency*). An invariant I of an object X *depends on a location* L if assignment to L might change I's value. Such a location L is called a *dependee* of I. The dependency is called *static* if L is an instance variable of X. It is called *dynamic* if L is a location or array element of a different object.

For example, the invariant of a `List` object X depends on X.`n`, X.`array`, X.`array.length`, and X.`array`$[i]$, for all i within the array bounds. The first two dependencies are static, the others are dynamic. More generally, invariant declarations refer to dependee locations by field or array *access expressions* of the form "`this.`$g_0 g_1 \ldots g_N$", for $N \geq 0$, where g_0 is a field name and each of the other g_i is either a field access of the form $.f$ or an array access of the form $[j]$. The leading "`this.`" can and will often be omitted in the following. Since we assume that there are no method calls in invariant declarations, the access expressions describe the dependencies of a class invariant. Dependencies with $N = 0$ are static. For dynamic dependencies ($N > 0$), g_0 is called the *pivot field* of the dependency.

Note that if X's invariant has a dynamic dependency on $X.g_0 \ldots g_N$, then all locations $X.g_0 \ldots g_k, k < N$ are dependees as well, since an assignment to $X.g_0 \ldots g_k$ redirects the dependency to a different location, which might have a different value.

3. The classical technique

In this section we explain the classical invariant technique. This technique is interesting both for its simplicity and for its limitations. These limitations are the problem we solve in later sections.

3.1. Classical encapsulation

The classical technique assumes a simple encapsulation discipline that does not permit layering. In this technique, invariants can only be guaranteed if their dependees are encapsulated or otherwise protected from unwanted modification. Dependees of the latter kind are fields that cannot be assigned directly by programs; in Java such fields include the `length` field of arrays and other final fields. We call such fields *constant fields*.

Definition 3.1 (*Classically Encapsulated*). A location $X.f$ is *classically encapsulated in object* X if the only assignments to $X.f$ occur in method executions with X as receiver object.

Classical encapsulation can be enforced by only allowing field assignments of the form `this.`f`=`E, as in Smalltalk [25] and Eiffel [6,7]. Fields of objects different from `this` have to be assigned via setter methods. However, note that this discipline allows methods in subclasses of the declaration class of f to modify f, that is, f need not be private.

3.2. Classical admissible invariants

For modular specification and verification, it is important to control dependencies. The key restriction of the classical approach is that dependencies on mutable fields must be static. That is, the access expressions permitted in the classical invariants of a class C are characterized by the following regular expression: $f_C\ c^*$, where f_C is a field declared in C and c is a constant field access.

Definition 3.2 (*Classical Invariant*). An access expression $g_0 \dots g_N$ occurring in a class C is *classically admissible*, if the field name g_0 is declared in C and each field access g_i, $0 < i \leq N$, is a constant field access.

An invariant declaration in class C is *classical* if each of its access expressions is classically admissible.

In a language like Java, Eiffel, or Smalltalk, where arrays are stored as separate objects on the heap, array element references are not classically admissible.[1] So the invariant of class `List` (Fig. 1) is not classical because the access expression `array[i]` is neither static nor does it refer to a constant field.

More generally, the classical technique does not permit layering of objects, because an invariant for an object cannot depend on the mutable state of other objects.

Orthogonal to the topic of invariants for layered object structures is the problem of invariants depending on superclass fields. Such dependencies cause a different modularity problem. By modifying superclass fields, code in the superclass could break subclass invariants without noticing it, because subclass invariants are usually not known in the superclass. To avoid this particular problem and to focus on the main topic, Definition 3.2 restricts class invariants so that they cannot depend on superclass fields. More precisely, invariants declared in a class C are only allowed to depend on non-constant fields that are declared in C. Similar restrictions with respect to superclass fields are necessary for soundness in other classical approaches [4,5]. In Section 11, we briefly discuss how techniques such as those proposed by Ruby and Leavens [26], and by Barnett et al. [12] might be adapted to address this restriction. So, for the bulk of this paper, while we do not ignore inheritance, we do not work on relaxing the restriction and instead concentrate on the classical technique's limited support for layered object structures.

3.3. Classical invariant semantics

The classical approach uses a visible state semantics [27].

Definition 3.3 (*Classical Invariant Semantics*). A program execution satisfies the *classical invariant semantics* if and only if each object satisfies its invariant in each visible state.

As pointed out at the beginning of Section 1.3, we cannot require that invariants hold in all execution states. Focusing on the visible states is probably the simplest way to solve the temporary violation problem: Invariants should hold at least in pre- and poststates and it is difficult to statically distinguish reasonable executions states in between.

The invariant semantics enforces that *each* object has to satisfy its invariant in *each* visible state. To illustrate the reason for this, we consider a call to a method m of class C with a parameter p of class D and assume that m calls method n on p. At the latter call, the invariant of D has to hold for p. In order to establish this invariant, it already has to hold in the prestate of the call to m. The same is true for all objects reachable from the receiver of m and from p. In particular, it does not suffice that all objects of the classes of the parameters satisfy their invariant.

3.4. Classical proof technique

Proof techniques formally describe how one shows correctness of a program with respect to its intended semantics. A proof technique is usually based on (1) proof obligations and (2) a program-independent *soundness proof* to show that the proof obligations are sufficient to verify that any program meeting those obligations satisfies the intended semantics. In a modular proof technique, proof obligations are imposed on modules independently of each other. A

[1] In a language, like Ada, where arrays are stored directly in objects and thus can be treated as shorthand for fields, array element references would be classically admissible. In C++ non-heap (direct) array references would be admissible, but heap-based (indirect) array references would not be.

modular proof technique that can be used to show correctness with respect to the classical invariant semantics is the following.

Definition 3.4 (*Classical Proof Technique*). Let C be a class and let $inv(C)$ denote the conjunction of the invariants declared in C and its superclasses. Let $pre(o.p(\vec{e}))$ and $post(o.p(\vec{e}))$ denote the pre- and postcondition of a call to method p, determined by the static types of o and $\vec{e}$.

For each method m declared in C one must show that:

(O1) $inv(C)$ and m's postcondition hold for the poststate of each execution of m, and
(O2) for every call of a method $o.p(\vec{e})$ that appears in the body of m, $pre(o.p(\vec{e}))$ holds in the call's prestate, and $inv(C)$ also holds for m's receiver object in the prestate of the call to p.

For the proof of each method m, one may assume that:

(A1) the invariants of all allocated objects and m's precondition hold in m's prestate, and
(A2) after each call of a method, $o.p(\vec{e})$, $post(o.p(\vec{e}))$ holds and all allocated objects satisfy their invariants.

Note that the above proof technique avoids the problem of re-entrant method executions (call-backs), pointed out by Huizing and Kuiper [28], by requiring in proof obligation (O2) that a method re-establishes the invariant of the current receiver object before it makes calls to other methods.

The classical proof technique can be applied in a modular way. To verify methods of a class C, one only has to refer to program parts that are available in C, such as C's superclasses or the specifications of methods that are called from methods of C. Therefore this technique enables modular reasoning.

Moreover, the classical proof technique is sufficient to show that classical invariants hold in all visible states. The proof of this soundness property is contained in Appendix A.

Theorem 3.5 (*Classical Soundness*). *Let* ***P*** *be a program containing only classical invariants. Suppose methods of* ***P*** *only assign to fields of* `this` *and array elements. If the classical proof obligations for* ***P****'s invariants hold, then all objects satisfy their invariants in all visible states.*

4. Limits of the classical technique

The restrictive limitations in the definition of classical invariants, which prevent layering, are needed for soundness. We next explain why these cannot be relaxed without either changing the notion of encapsulation or the proof technique.

4.1. Abstraction layering is not sound

The classical proof technique cannot be soundly generalized to invariants that depend on mutable objects of underlying layers. Consider the class `BagWithMax` from Fig. 2 again. According to the classical technique it would be perfectly fine to omit the last statement of `BagWithMax`'s `insert` method! Why? Because just before that statement is a call to a method, `theList.add(k)`, and since that call's poststate is a visible state, the invariants of all allocated objects must hold. Since the receiver (`this`) in `BagWithMax`'s `insert` method is certainly an allocated object in this state, its invariant must hold, according to the classical technique. But that is not true if `k` is greater than `maxElem` (hence the need for the last statement). This shows that relaxing the restrictions on invariants leads to an unsoundness.

Another way to view this soundness problem is that the classical invariant semantics is too strong for invariants over layered object structures. The class `List` cannot modularly know enough to establish the invariant of a class, `BagWithMax`, that it does not know about.

4.2. Mutable subobjects are not sound

In the classical technique, the invariant of an object X may only depend on those fields that can exclusively be assigned to by methods executed on X. In particular, it must not depend on mutable fields of objects that are referenced by X, such as subobjects in lower abstraction layers used for X's implementation. This restriction is necessary as long as there is no control of aliasing. The need for alias control is illustrated by the `List` example (Fig. 1). `List`'s invariant

```
/*@ requires elems != null
  @     && (\forall int i; 0<=i && i<elems.length; elems[i]>=0);
  @*/
public void addElems(int[] elems) {
  if (n==0) {
    array = elems;     // illegal
    n = elems.length;
  } else {
    /* ... */
  }
}
```

Fig. 3. A questionable implementation of the `List` method `addElems` that stores the argument array into the `array` field. Our approach will require copying `elems` (or some form of ownership transfer).

```
class Client {
  /*@ requires list.n == 0; @*/
  void violator(List list) {
    // invariant of list holds in prestate
    int[] aliasedArray = new int[10];
    list.addElems(aliasedArray);
    aliasedArray[0] = -1;
    // invariant of list is violated in poststate
  }
}
```

Fig. 4. Client code that shows the problem with aliased representations.

depends on its array elements. Methods that get hold of the reference to this array object could assign to its elements and violate `List`'s invariant. Such an alias could occur by rep exposure [10] or by capturing [29] as illustrated by the version of `addElems` in Fig. 3. With that version, the code fragment in Fig. 4 would violate `List`'s invariant. This scenario shows that a sound technique must either restrict invariants that depend on subobjects in lower abstraction layers, or it must control modifications of such subobjects.

4.3. Summary of the technique's problems

In sum, the classical technique effectively cannot describe interesting invariants for layered objects, unless the subobjects are immutable. The restrictive limitations of classical invariants stem from the visible state semantics and from the classical proof technique. As we have shown by example, to remove these limiting restrictions, and hence to support invariants for interesting layered object structures, it is necessary to change either the visible state semantics or the proof technique, or both. The challenge is to do that and still preserve modularity.

5. Background on ownership

Our first solution technique builds on an ownership model. An ownership model partitions the allocated objects into a hierarchy of disjoint groups of objects, called "contexts" [9,30–35]. Contexts are used:

(1) for alias control and encapsulation (preventing representation exposure);
(2) to control the dependencies of invariants (preventing argument exposure [10]);
(3) to define a weaker semantics for invariants that allows layering.

The first two aspects allow one to encapsulate objects even in layered object structures. The third aspect is vital to our extended proof technique's soundness. The notions and properties of the ownership model that we need in the rest of this paper are introduced in two steps. First, we describe the additional structure that the ownership model provides. Second, we explain how ownership is specified and enforced.

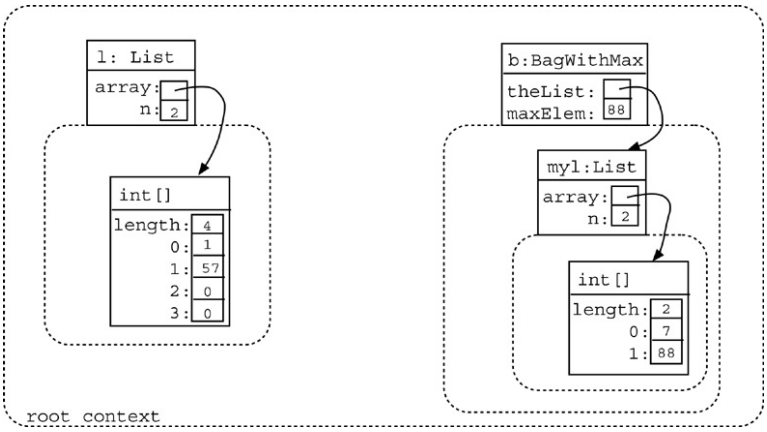

Fig. 5. A hierarchy of contexts (rounded rectangles, dotted lines) with the objects (rectangles) they contain. Owner objects sit atop the context they own. In general, a context can contain many objects, several of which may not be directly referenced by their owner.

5.1. Ownership model

Within the ownership model, each object is directly owned by at most one other object, called its *direct owner*, or *owner* for short. A *context* is the set of all objects with the same direct owner. The set of objects without an owner is called the *root context*. The owner-relation is required to be acyclic. Consequently, the contexts form a tree with the root context as root and where context Δ is a child of context Γ if and only if the owner of the objects in Δ is an element of Γ. The owner of an object is specified when the object is created and cannot change later.

We say that object X is the owner of context Γ if X is the direct owner of the elements of Γ. Moreover, we say that an object Y is *inside* a context Γ if it is an element of Γ or of one of its descendants; otherwise Y is *outside* Γ. X *owns* an object Y if Y is inside the context owned by X. For instance, in Fig. 5, `myl` is an element of the context owned by `b`. Hence, it is inside the root context, but not an element of the root context, and it is outside the context owned by the other `List` object, `l`. Finally, a *method is executed in a context* Γ if and only if the corresponding receiver object is an element of Γ.

We use the ownership model to restrict references between objects. An object X can have a direct reference to objects in the same context as X and to objects directly owned by X. That is, the only references that cross context boundaries, i.e., references going from an object in one context to an object in a different context, are references from the owner of a context Γ to elements of Γ. As the restrictions of this core model do not allow the use of several important programming patterns, we will relax them in Section 7 by allowing special readonly references to point to objects in arbitrary contexts. The restrictions apply to references stored in all locations, including local variables and formal parameters. Technically, local variables and formal parameters in a method m are treated like fields of m's receiver object.

5.2. Enforcing the ownership model

Several type systems have been proposed to statically enforce ownership models [30–35]. The work presented here builds on common properties of almost all ownership type systems. Therefore, we expect that progress in the theory of ownership type systems can be directly translated into less restrictive versions of our work. For concreteness we use the Universe type system, which has been implemented in JML [36].

Types. Our ownership model permits (1) references between objects in the same context (*peer* references) and (2) references from an object X to an object directly owned by X (*rep* references).[2] This classification is expressed in the Universe type system by adding one of the ownership modifiers `peer` and `rep` to each reference type. The `peer` modifier is the default modifier, that is, it can be omitted. For instance, the type `/*@ rep @*/ T` is the type of references pointing to objects of class `T` owned by `this`. Rep and peer types are disjoint, that is, references of type

[2] We relax this restriction by adding readonly references, which are allowed to cross context boundaries, in Section 7.

Universe Type Annotations			
Case	X	f	$X.f$
1.	`peer`	`peer`	`peer`
2.	`peer`, $X =$ `this`	`rep`	`rep`
3.	`rep`	`peer`	`rep`
4.	`peer`, $X \neq$ `this`	`rep`	*error*
	`rep`	`rep`	*error*

Fig. 6. Summary combinations for field access.

`/*@ rep @*/ T` cannot be assigned to variables of type `/*@ peer @*/ T` and vice versa. This means that objects of rep and peer types are elements of different contexts.

Figs. 1 and 2 demonstrate the use of ownership modifiers: A `List` object owns its `array` object, and a `BagWithMax` object owns its `theList` object. This ownership relation is illustrated by Fig. 5, which shows the root context with a `List` and a `BagWithMax` object and their child contexts. In Fig. 1, the `elems` parameter of `List`'s method `addElems` has an implicit `peer` modifier, which indicates that the argument array has to be in the same context as the receiver object of the method.

The owner of a new object is determined by the type given in the creation expression: `new /*@ rep @*/ T(...)` creates an object in the context owned by `this` (as in Fig. 1), whereas `new T(...)` (or the more explicit `new /*@ peer @*/ T(...)`) creates an object in the same context as `this`. In the latter case, `this` and the new object either belong to the root context or have the same owner (see Section 5.1).

Type rules. The type rules of the Universe type system guarantee that the ownership modifiers of reference types correctly reflect the owner of the referenced object. We summarize the most important rules in the following. For a formalization of the complete type system, see our earlier work [3,35].

Assignment: The rule for an assignment is the standard Java rule: The type of the right-hand side expression has to be a subtype of the type of the left-hand side variable. This rule renders the implementation of method `addElems` in Fig. 3 incorrect: The type of `elems`, `/*@ peer @*/ int[]`, is not a subtype of the type of `array`, `/*@ rep @*/ int[]`.

Field access: To determine the owner of an object referenced by $X.f$ – and, thus, the type of the field access $X.f$ – one has to consider the ownership modifiers of the types of both X and f. The possible *combinations* are as follows (see Fig. 6):

(1) If the types of both X and f are peer types, then we know (a) that the object referenced by X has the same owner as `this`, and (b) that the object referenced by $X.f$ has the same owner as X and, thus, the same owner as `this`. Consequently, the type of $X.f$ also has the modifier `peer`.
(2) If the type of f is a rep type, then the type of `this`.f has the modifier `rep`, because the object referenced by `this`.f is owned by `this`.
(3) If the type of X is a rep type and the type of f is a peer type, then the type of $X.f$ has the modifier `rep`, because (a) the object referenced by X is owned by `this`, and (b) the object referenced by $X.f$ has the same owner as X, that is, `this`.
(4) In all other cases, we cannot determine statically that the object referenced by $X.f$ has the same owner as `this` or is owned by `this`. Therefore, these forms of field accesses are forbidden and lead to a type error.

The rules for array access are analogous to field access.

Method call: Analogously to field accesses, the declared parameter and result types of a method have to be interpreted w.r.t. the type of the receiver expression of the method call. Consider a method `foo(/*@ peer @*/ T p)`. The `peer` modifier expresses that the parameter `p` has the same owner as the receiver object on which `foo` is executed. Therefore, if `foo` is called on an expression of a rep type, the actual parameter of the call must also be of a rep type to meet this requirement. This interpretation of parameter and result types is performed by the mapping described for field accesses, with the type of the field replaced by the parameter or result type of the method. Thus, in our example, the combination of a rep type (the type of the receiver) and a peer type (the type of `p`) yields a rep type (point (3) in the enumeration above).

For a call $X.m(\ldots a_i \ldots)$, this observation leads to the following conditions: (1) The type of each actual parameter a_i must be a subtype of the combination of the types of X and of the corresponding formal parameter p_i. (2) The type of the call expression is the combination of the types of X and the declared result type of m.

5.3. Properties of the ownership model

The Universe type system guarantees the following program invariant.

Theorem 5.1 (*Universe Invariant*). *Let **P** be a program that is type correct in the Universe type system. The following program invariant holds in every execution state: If an object X holds a direct reference to an object, Y, then* (1) *X and Y are elements of the same context or* (2) *X is the owner of Y. W.r.t. this program invariant, local variables and formal parameters behave like instance variables of the* `this` *object.*

Since the Universe type system is not a contribution of this paper, we refer to Müller's thesis for the type safety proof [3].

The Universe invariant implies the following properties of our ownership model.

Corollary 5.1 (*Ownership Properties*). (1) Ingoing reference invariant*: Every reference chain from an object outside a context Γ to an object inside Γ passes through Γ's owner.*
(2) Outgoing reference invariant*: There is no reference chain from an object inside a context Γ to an object outside Γ.*
(3) Context encapsulation*: Method executions in a context Γ can only be invoked by another method execution in Γ or by method executions with the owner of Γ as* `this` *object. Elements of arrays in Γ can only be assigned to by method executions in Γ or by method executions with the owner of Γ as* `this` *object.*
(4) Context locality*: Method executions in a context Γ can only modify locations of objects inside Γ, either directly by field updates and array element assignments, or indirectly by method invocations.*

Properties (1) and (2) are immediate consequences of the Universe invariant. Properties (3) and (4) follow from Properties (1) and (2). Properties (1) and (3) guarantee that an owner has full control over the objects in its context. They generalize classical encapsulation of fields of the `this` object to encapsulation of the fields of all objects owned by `this`. Property (2) will be used to restrict dependencies. Property (4) allows one to localize effects of method executions.

It is helpful to also look at the context encapsulation property from the perspective of the method performing an invocation or an update. This leads to the following equivalent property.

Corollary 5.2 (*Alternative Context Encapsulation Property*). *A method executed in context Δ can only invoke other methods in context Δ or in the context owned by the* `this` *object. Analogously, this method execution can only assign to fields and elements of arrays in these two contexts.*

The classes in Fig. 7 illustrate how the Universe type system guarantees the ownership properties in Corollary 5.1. Consider the object structure shown in Fig. 8: two instances of class `Superclass`, `s1` and `s2`, own `T` objects `t1` and `t2`, respectively.

The statements in method `bad` are illegal because they potentially violate the ownership properties. Consider the call `s1.bad(s2)`.

Statement (1) violates the ingoing reference invariant because it creates a direct reference from `s1` into the context owned by `s2`. The assignment is illegal because `superT` has a rep type and the receiver, `s`, is different from `this`. Therefore, cases 1 to 3 of the rule for field accesses do not apply. For the same reason, statement (2) is illegal. Note that the `leak` method itself is permitted, but the rule for method calls ensures that this method can be called only on the receiver `this` because it returns a rep reference.

Statement (3) violates the outgoing reference invariant by creating an outgoing reference from `t1` to `s1`. The combination of the types of the receiver, `/*@ rep @*/ Superclass`, and the formal parameter, `/*@ peer @*/ Superclass`, yields `/*@ rep @*/ Superclass`. Therefore, the call is illegal because the method expects an argument of type `/*@ rep @*/ Superclass`, but the actual parameter is of type `/*@ peer @*/ Superclass`.

```
class T {
  private /*@ spec_public peer @*/ Superclass out;

  public void setOut(/*@ peer @*/ Superclass o) {
    out = o;
  }
}

class Superclass {
  private /*@ rep @*/ T superT;

  private /*@ rep @*/ T leak() {
    return superT;
  }

  public void bad(/*@ peer @*/ Superclass s) {
    superT = s.superT;    // (1) illegal
    superT = s.leak();    // (2) illegal
    superT.setOut(this);  // (3) illegal
  }

  public void init() {
    superT.setOut(null);
  }
}
```

Fig. 7. Classes illustrating how the Universe type system enforces the ownership properties. Method `bad` violates the type rules for field accesses and method calls.

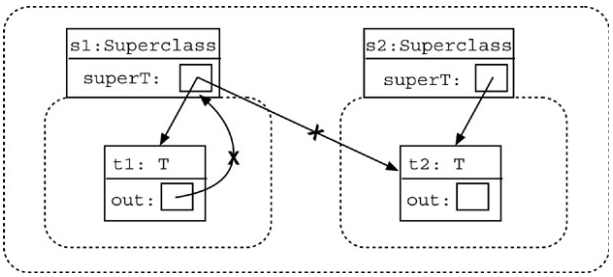

Fig. 8. Ownership structure for the example in Fig. 7. The ownership properties guarantee that `s1` and `s2` reference different `T` objects. The crossed-out references are those references that would be introduced by the illegal statements in Fig. 7.

5.4. Subclass separation

The ownership model allows an object X to control how objects inside the context owned by X are modified. In the next section, we will use this property to allow X's invariant to depend on the state of objects owned by X. Class `Subclass` in Fig. 9 contains such an invariant: for a `Subclass` object X this invariant requires that the `out` field of the `T` object X`.subT` is non-null.

Although X`.subT` is owned by X and X can control how that field is modified, `Subclass`'s invariant leads to a potential soundness problem. Since class `Superclass` does not declare an invariant, method `init` can be verified easily. However, if X`.superT` and X`.subT` both reference the same `T` object Y, then calling X`.init` violates `Subclass`'s invariant by setting Y`.out` to `null`.

To avoid this source of unsoundness, we enforce that the set of objects reachable via a superclass rep field and the set of objects reachable via a subclass rep field are disjoint. (Again, formal parameters and local variables of a method in class C behave like fields of `this`, declared in C.) This is achieved by requiring that `rep` fields, and methods that

```
class Subclass extends Superclass {
  private /*@ spec_public rep @*/ T subT;

  //@ public invariant subT != null && subT.out != null;

  public void separate() {
    subT = superT;  // illegal
    subT = leak();  // illegal
  }

  // other methods and constructors omitted.
}
```

Fig. 9. The invariant of class `Subclass` is violated by `Superclass`'s `init` method if `superT` and `subT` reference the same `T` object. Such an alias is forbidden.

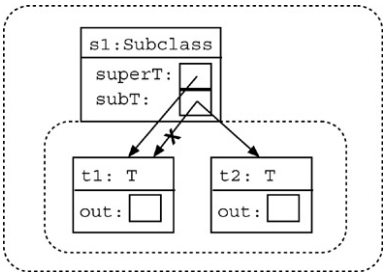

Fig. 10. Ownership structure for the example in Fig. 9. Subclass separation guarantees that `superT` and `subT` reference different `T` objects. The crossed-out reference would be introduced by the illegal statements in Fig. 9.

have `rep` modifiers for their parameters or results, are private. Therefore, a rep reference stored in a subclass field cannot be passed to superclass methods. This restriction renders the statements in `Subclass`'s method `separate` illegal since they access a private field and a private method of the superclass.

The requirement that `rep` fields, and methods that have `rep` modifiers for their parameters or results, are private guarantees the following additional ownership property. It is illustrated by Fig. 10.

Corollary 5.3 (*Subclass Separation*). (5) Subclass separation: *Let X be the owner of a context Γ, f a field of X, and m a method of X. If f and m are declared in different classes, an execution of m on receiver X cannot modify objects inside Γ that are reachable from $X.f$.*

Method `separate` in class `Subclass` illustrates how subclass separation is enforced. The method cannot create an alias between the superclass field `superT` and the subclass field `subT` by a direct field access or by calling `leak` because both `superT` and `leak` must be private, since they have rep types.

6. The ownership technique

The ownership model allows one to encapsulate whole object structures as underlying layers in the representations of owner objects. In this section, we exploit this property to generalize the classical technique to invariants over layered object structures. To avoid the soundness problems described in Section 4, we use the hierarchical structure of the ownership model to refine the classical invariant semantics and also present a refined proof technique.

6.1. Ownership encapsulation

In the ownership model, objects outside a context Γ can modify objects inside Γ only by invoking methods of Γ's owner object X. So we can consider all objects in Γ to be part of X's encapsulated representation.

Definition 6.1 (*Ownership Encapsulated*). A field location $Y.f$ or an array location $Y[i]$ is *ownership encapsulated in an object* X if:

(1) $X = Y$ and the only assignments to the field location $X.f$ occur in method executions with X as receiver object, or
(2) X owns Y.

Part 1 above corresponds to the definition of classical encapsulation (Definition 3.1). It is enforced in the same way, by disallowing assignments to fields of objects other than `this`. Part 2 enables owned objects (including arrays) as part of the encapsulated representation. It is enforced by the Universe type system.

Although owned objects are encapsulated in their owner, we do not allow the owner to directly assign to fields of owned objects. As in the classical technique, method calls must be used to change fields of other objects. While this restriction could be relaxed [3], it simplifies the proof obligations for invariants significantly.

6.2. Ownership admissible invariants

Using ownership encapsulation allows the key generalization of the classical technique necessary to support layered object structures. That is, in the ownership technique, an invariant of an object X may depend on mutable locations of objects that X owns. That is, the ownership admissible access expressions that can appear in a class C are characterized by the following regular expression: $(f_C \mid r_C g^*)\, c^*$, where f is a field, r is a `rep` field, g is a non-constant field or array access, c denotes a constant field access, and the subscript C indicates that the field is declared in class C.

Definition 6.2 (*Ownership Admissible*). Let $E = g_0 \dots g_N$ be an access expression occurring in a class C and M, $0 \le M \le N$, be the minimal index such that for each i, $M < i \le N$, g_i is a constant field access. Then E is *ownership admissible* if the field name g_0 is declared in C and $g_0 \dots g_M$ matches one of the following cases:

(1) $M = 0$, or
(2) $M > 0$ and the pivot field g_0 is of a `rep` type.

An invariant declaration in class C is *ownership admissible* if each of its access expressions is ownership admissible.

Access expressions of form (1) correspond to the classical approach's static dependees (Definition 3.2). For the invariant of object X, access expressions of form (2) denote locations of objects inside the context owned by X; these dependees were not allowed in classical encapsulation. Recall that the `rep` annotation indicates that the object referenced by $X.g_0$ and, thus, all objects reachable from $X.g_0$ via peer and rep references are in contexts owned by X (due to the outgoing reference invariant, Corollary 5.1).

As an example, we consider the invariant of class `List` with its access expressions `n`, `array.length`, and `array[i]`. Since `length` is a constant field, both `n` and `array.length` have form (1). The expression `array[i]` matches form (2), because the array elements are non-constant and `array` is a `rep` pivot field.

Ownership admissible invariants can express properties of layered object structures as long as the representation is encapsulated in a context. For instance, `BagWithMax`'s invariant is allowed to depend on locations of the associated list, because the ownership model guarantees that all modifications of the list are initiated by a method of the owning `BagWithMax` object, and this `BagWithMax` method makes sure that the invariant is preserved. In particular, invariant violations through representation exposure, as described in Section 4, are ruled out by the ownership model.

Due to our assumed syntax for invariants, in particular, the absence of method calls, a type checker can syntactically determine the dependees of an invariant and can statically check admissibility of invariants.

6.3. Relevant invariant semantics

In Section 4, we showed that invariants over layered object structures may not hold in all visible states. The reason is that an object needs the right to temporarily violate its own invariant when changing an encapsulated object. For example, a `BagWithMax` object needs to temporarily violate its invariant when changing its encapsulated `theList` object. The difference from the visible state semantics is that these invariant violations can span method executions in the owned context. For example, a `BagWithMax` object can violate its invariant during a call to a method of `theList`,

since `theList` is owned by the `BagWithMax` object. To allow such violations, we weaken the visible state semantics from the classical technique (Definition 3.3).

The basic idea of the weakened semantics is that methods executing in a context Γ are regarded as internal operations that might break the invariants of objects in contexts that are ancestors of Γ. The rationale is that (1) according to context locality (Corollary 5.1), such method executions cannot modify objects outside Γ, and (2) according to context encapsulation, such method executions are controlled by a method execution of Γ's owner. Γ's owner is responsible for invariants of objects in its context.

Definition 6.3 (*Relevant Object*). An object X is *relevant to the execution of a method m* if and only if X is inside the context in which m executes.

For example, in the execution of method `add` on object `myl` of Fig. 5, the relevant objects are those inside the context that contains `myl` (here `myl` and the array owned by `myl`). Since the field `array` in `List` is declared using the `rep` annotation, the object that `myl.array` points to is owned by `myl`. Thus, this array is also relevant. However, the `BagWithMax` object that owns `myl` is not relevant.

Definition 6.4 (*Relevant Invariant Semantics*). A program execution satisfies the *relevant invariant semantics* if for each execution of a method m, and for each object X relevant to m's execution, X satisfies its invariant in the visible states of m's execution.

For simplicity, if object X is relevant to the execution of method m, we refer to X's invariant as being *relevant to the execution of m*. We use the term *relevant invariant* to refer to such invariants when the method execution is clear.

For example, consider the execution of method `add` on object `myl`. For this execution, the invariant of `myl` is relevant, but the invariant of the `BagWithMax` object that owns `myl` is not relevant.

6.4. Ownership proof technique

The ownership proof technique describes how to prove correctness of invariants with respect to the relevant invariant semantics. The classical proof technique (Definition 3.4) is not sound for use with the relevant invariant semantics, as its assumption that all invariants hold in all visible states is too strong. Furthermore, we can also weaken the proof obligations of the classical technique, due to ownership encapsulation. Changes from the classical technique are highlighted in bold in the following definition.

Definition 6.5 (*Ownership Proof Technique*). Let C be a class and let $inv(C)$ denote the conjunction of the invariants declared in C and its superclasses. Let $pre(o.p(\vec{e}))$ and $post(o.p(\vec{e}))$ denote the pre- and postcondition of a call to method p, determined by the static types of o and $\vec{e}$.

For each method m declared in C one must show that:

(O1) $inv(C)$ and m's postcondition hold for the current receiver object in each poststate of m's execution, and
(O2) for every call of a method $o.p(\vec{e})$ that appears in the body of m, $pre(o.p(\vec{e}))$ holds in the call's prestate, and **if** o **is an element of** m**'s context, then** $inv(C)$ also holds for m's receiver object in the prestate of the call to p.

For the proof of each method m, one may assume that:

(A1) the invariants of all allocated objects **that are relevant to the execution of** m and m's precondition hold in m's prestate, and
(A2) after each call of a method $o.p(\vec{e})$, $post(o.p(\vec{e}))$ holds and all allocated objects **that are relevant to the execution of** $o.p(\vec{e})$ satisfy their invariants.

Since ownership encapsulation prohibits assignment from outside an object to fields of that object, these proof obligations can be verified modularly (see Section 3.4). Note that obligation (O2), which is necessary to avoid problems with re-entrant method calls, is weaker than for the classical technique. This is because the outgoing reference invariant of the ownership model prevents re-entrant calls on objects outside the context that contains the current receiver object, `this`. That is, if p is executed on an object owned by `this`, the execution of p cannot re-enter the `this` object by a call-back. For this reason, the slightly weaker assumption of proof obligation (O2) suffices for soundness.

To illustrate how to use the ownership proof technique, consider BagWithMax's insert method. For the call to List's add method we may, by (A2), assume that the method preserves the invariant of theList. However, since the BagWithMax object this is not relevant to this call to add, its invariant might be broken by the call. To show that the insert method preserves this invariant (O2), we use the postcondition of add (A2) to derive that the list after the call contains exactly the elements before the call plus the new element k. If k happens to be a new maximum in the list, then BagWithMax's invariant is violated after the call, but re-established by the subsequent assignment to maxElem, which satisfies (O1).

As can be seen from the example above, responsibility for verifying invariants is divided. A method's implementor is responsible for the objects that are relevant to its executions. The clients of the method might be responsible for further objects. Which invariants are relevant is decided statically using the Universe type system. There are only two possibilities:

(1) The receiver expression of the call is of a peer type. That is, the caller and receiver of an invocation are in the same context, and, thus, the same invariants are relevant for the calling and the called method. Therefore, the guarantees provided by the called method imply the obligations the calling method has to satisfy.
(2) The receiver expression of the call is of a rep type. In this case, the receiver is owned by the caller Z. As illustrated by the BagWithMax example above, the call can violate the invariant of Z, because Z is not relevant to the call. However, the invariant of Z can and must be re-established in the remaining part of the calling method. That this is actually done can be verified using the specification of the called method.

This proof technique, besides handling layers of mutable objects and supporting modular verification, is sound with respect to the relevant invariant semantics. The proof of the soundness theorem below is contained in Appendix B.

Theorem 6.6 (*Ownership Invariant Soundness*). *Let* ***P*** *be a program that is type correct in the Universe type system. Suppose all invariants of* ***P*** *are ownership admissible and methods of* ***P*** *only assign to fields of* this *and array elements.*

If all invariants of ***P*** *have been proved using the ownership proof technique, and if X is an object relevant to the execution of a method m, then X satisfies its invariant in the visible states of m's execution.*

7. Readonly references

An important extension to ownership type systems, and one that relaxes some of the restrictions of the ownership technique, is *transitive readonly references* [37,35,38]. Such references can cross context boundaries, thereby permitting patterns that would otherwise violate the ingoing and outgoing reference invariant (properties (1) and (2) of Corollary 5.1). Among such patterns are collections that do not own their elements, iterators, binary methods, and clone operations. Transitive readonly references can be allowed to cross contexts since they cannot be used to modify the referenced objects. Both the ownership technique, and the visibility technique (described below) are easily extended to cover such readonly references.

In this section we describe how to extend the ownership technique to cover transitive readonly references. The main semantic complication is a special proof obligation for certain calls of side-effect free methods.

7.1. Syntax

The basic mechanism is simple. Besides the ownership modifiers rep and peer, we add the modifier readonly to the annotation language. Variables and expressions with this modifier can hold references to objects in arbitrary contexts. However, such *readonly references* must not be used to modify the referenced object. In JML, this restriction is checked syntactically: a readonly expression must not be used as receiver expression of a field update, an array update, or an invocation of a non-pure method. Pure methods are side-effect free methods. We allow pure methods to be called on readonly receivers because they do not modify the state of the receiver (or any other objects).

Class ListIterator (Fig. 11) illustrates how readonly references can be used to implement iterators. A List object and the associated iterators are in the same context. The iterator stores a reference to the array of the list. Since this array is owned by the List object, this reference has to be readonly as indicated by the readonly modifier of field theArray. Method getNext uses the readonly reference to directly access the internal list representation. However, the readonly reference to the array cannot be used to modify the list. Modifications must be performed by the owning

```
class ListIterator {
  private /*@ spec_public @*/ List theList;
  private /*@ spec_public readonly @*/ int[] theArray;
  private /*@ spec_public @*/ int position;

  /*@ public invariant theList != null
    @           && theList.array == theArray
    @           && 0 <= position
    @           && position <= theList.n;
    @*/
  public /*@ pure @*/ boolean hasNext() {
    return position != theList.size();
  }

  public int getNext() {
    return theArray[position++];
  }

  // other methods and constructors omitted.
}
```

Fig. 11. `ListIterator` implements iterators for the array-based `List` (Fig. 1). The iterator has a direct readonly reference to the list's internal array. `ListIterator`'s invariant is not admissible in the ownership technique because `theList` has no `rep` annotation.

```
class Purse {
  private /*@ spec_public @*/ int amount;

  //@ public invariant 0 <= amount;

  //@ ensures 0 <= \result;
  /*@ pure @*/ int getAmount() {
    return amount;
  }

  // other methods and constructors omitted.
}
```

Fig. 12. The pure method `getAmount` of class `Purse` can be called on readonly receivers.

`List` object. `ListIterator` objects store a reference to the associated `List` object, which can be used to delegate modifications to the list (not shown in the example).

In the Universe type system, readonly types are supertypes of the corresponding peer and rep types. Therefore, expressions of peer and rep types can be assigned to variables of readonly types. Conversely, downcasts can be used to cast readonly into rep or peer types. Runtime checks guarantee that the more specific owner information of the rep or peer type is correct and that subclass separation is preserved [13].

Readonly types are used for all expressions that cannot be statically determined to evaluate to a peer or a rep reference. That is, instead of forbidding expressions of form (4) in the Universe type rule for field accesses (Section 5.2), these expressions are typed readonly. For instance, if `ro` is a variable of a readonly type and field `f` has type `peer T`, then `ro.f` has type `readonly T` because the expression evaluates to a reference to a `T` object in a context that is not known statically.

Readonly references and pure methods are illustrated by the classes `Purse` and `Person` (Figs. 12 and 13). Method `getAmount` of class `Purse` relies on the invariant to guarantee its postcondition. A person can spend their money and the money of their spouse. `Person`'s method `amountSpendable` accesses the `Purse` objects of `this` and of `spouse`

```
class Person {
  private /*@ spec_public @*/ Person spouse;
  private /*@ spec_public rep @*/ Purse purse;

  //@ public invariant purse != null && spouse != this;

  //@ ensures 0 <= \result;
  public /*@ pure @*/ int amountSpendable() {
    int res = purse.getAmount();
    if (spouse != null) {
      /*@ readonly @*/ Purse p = spouse.purse;
      res += p.getAmount();
    }
    return res;
  }

  // other methods and constructors omitted.
}
```

Fig. 13. Method `amountSpendable` of class `Person` accesses the `Purse` object owned by `spouse`. Since this object is neither in the same context as `this` nor owned by `this`, the field access `spouse.purse` yields a readonly reference.

to determine the amount of money the person can spend. The field access `spouse.purse` yields a readonly reference since the referenced `Purse` object is owned by `spouse` and is, thus, neither in the same context as `this` nor owned by `this`. Method `amountSpendable` can use this readonly reference to call the pure method `getAmount`, but could not modify the spouse's `Purse` object, for instance, to take out money.

An ownership model with readonly references guarantees a slightly weaker program invariant and, therefore, ownership properties that are weaker than those in Corollary 5.1. The ingoing and outgoing reference invariants hold for chains of read–write (that is, peer or rep) references, but not for reference chains that contain readonly references. Context encapsulation and Corollary 5.2 hold for non-pure methods, but pure methods can be invoked from any context via a readonly reference. Context locality and subclass separation (Corollary 5.3) are not affected by readonly references.

Whereas the weaker ownership properties provide additional ways to read an object's state, they do not enable additional modifications. This allows us to soundly adapt the ownership technique to these weaker ownership properties, as described next.

7.2. Ownership technique with readonly references

When combined with readonly references, the ownership technique can handle even more examples including collections of object identities, such as object lists, iterators, etc. In this subsection, we explain how the technique described above can be extended to an ownership model with readonly references.

To maintain the encapsulation principle, an object needs control over the locations its invariant depends on. Therefore, the invariant of an object X may only depend on mutable locations of X and objects owned by X. This is guaranteed by requiring that the fields or array elements $g_1, \ldots, g_{N-1}$ in form (2) of the definition of admissible invariants (Definition 6.2) are not declared with a `readonly` annotation. That is, we refine case (2) to:

(2) $M > 0$, *the pivot field* g_0 *is of a* `rep` *type,* ***and all fields and array components in*** $g_1, \ldots, g_{M-1}$ ***are of a*** `rep` ***or*** `peer` ***type.***

The proof obligations of the ownership technique (Definition 6.5) stay essentially unchanged. However, calls of pure methods on readonly receivers impose an additional obligation. When a readonly reference is used as receiver of a method call, the receiver X can be in an arbitrary context. In particular, X can be one of the (transitive) owners of `this`, whose invariant is temporarily violated. That is, the method may be executed in a state in which the relevant object X does not satisfy its invariant, which clearly violates the relevant invariant semantics (Definition 6.4).

To avoid the unsoundness caused by such re-entrant calls, we add the following proof obligation to the proof obligations of the ownership technique (Definition 6.5):

(O3) *For each method m, and for each method call $o.p(\vec{e})$, appearing in m, if o has a readonly type, then one must show that the object referenced by o is inside the context that contains m's* `this` *object.*

Since the receiver o of a call to method p is inside the context of the caller, all objects relevant to the execution of p are also relevant to the caller. Therefore, the invariants of these objects hold in the prestate of p. Due to this proof obligation only ingoing readonly references can be used as receivers of method calls.[3] However, outgoing readonly references can still be used to compare object identities or read fields.

In many practical cases, the additional proof obligation can be shown by using Universe type information. Consider the call `p.getAmount()` in `Person`'s `amountSpendable` method. We know that `p` is inside the context that contains the `this` object, because (1) `spouse` is in the same context as `this` (`spouse` is a peer field) and (2) `spouse.purse` is in the context owned by `spouse` (`purse` is a rep field). Consequently, `spouse.purse` and, thus, `p` is inside the context that contains the `this` object.

In the `ListIterator` example, the invariant and the types of `theList` and `List`'s `array` field can be used to show that the array referenced through `theArray` is inside the context that contains the `ListIterator` object. Again, this allows `ListIterator` to call pure methods on the array, for instance, `equals` (not shown in the example).

The ownership technique in the presence of readonly references is sound, that is, with the adapted proof obligations, Theorem 6.6 still holds. For non-pure methods m, the proof in Appendix B remains valid. For calls of pure methods, the additional proof obligation guarantees that all relevant objects satisfy their invariant in the prestate of m's execution. Since m is side-effect free, it cannot violate any invariants. Thus, the relevant invariants still hold in the poststate.

8. Limitations of the ownership technique

The ownership technique presented above is a proper generalization of the classical technique for reasoning about invariants. This is because a Java program that meets the assumptions of the classical approach (Definitions 3.2 and 3.1), trivially passes the Universe type system (because it has no `rep` and `readonly` annotations) and meets the assumptions of the ownership technique (Definitions 6.1 and 6.2).

In addition to all implementations that can be handled by the classical approach, the ownership technique is capable of expressing invariants of non-trivial layered object structures that are encapsulated, that is, those that are accessed through a single owner object. Such structures include record-like collections of data (e.g., `Person` objects) and recursive data structures such as balanced trees that are accessed through the root object.

However, ownership encapsulation (Definition 6.1), even with the addition of transitive readonly references, is too strong for several interesting implementations. In addition to the restrictions imposed by the ownership model (Section 5.2), the ownership technique has the following limitations.

Mutually dependent peers: Invariants over mutually recursive or cyclic data structures can only be handled if the mutually dependent objects are encapsulated and controlled by an owner. In this case, the invariant can be specified in the class of the owner. Sometimes objects of a class C are mutually dependent without having a natural owner. If we specify the corresponding invariant in C, it is not ownership admissible. For instance, the additional invariant of class `Person` in Fig. 14 is not admissible in the ownership technique since a person does not own its spouse. This problem cannot be fixed by declaring `spouse` as `rep`, because that would mean that X`.spouse` would have a rep reference to its owner X (since X`==` X`.spouse.spouse`), which is forbidden by the outgoing reference invariant (Corollary 5.1).

Direct field access: A method can only assign to fields of the `this` object. As shown by method `marry` (Fig. 14), it is often convenient to allow cooperating objects to directly assign to each other's fields.

Iterators: Iterators have invariants that depend on the states of the collections over which they iterate. For example, the `ListIterator` in Fig. 11 depends on `theList.n`. However, to allow several iterators for one list and to avoid copying the list, the iterator does not own the list (`theList` is not declared with `rep`). Such invariants are not ownership admissible.

[3] On outgoing readonly references, only pure helper methods, which do not assume invariants to hold, could be called.

```
class Person {
  private /*@ spec_public @*/ Person spouse;
  // ... other members and specifcations as before

  //@ public invariant spouse != null ==> spouse.spouse == this;

  /*@ requires p != null && p != this
    @  && spouse == null && p.spouse == null;
    @*/
  public void marry(Person p) {
    spouse = p;
    p.spouse = this;
  }
}
```

Fig. 14. The extension of class `Person` (Fig. 13) is not admissible in the ownership technique because `spouse` has no `rep` annotation and method `marry` directly assigns to `p`'s field.

These three limitations stem from the principle that the invariant of an object X can depend only on those locations whose modification X can control. In the next section, we describe how to achieve more expressiveness for the price of more proof obligations.

9. The visibility technique

The visibility technique, presented in this section, directly addresses the problems of mutually dependent peers, direct field access, and iterators that were described in Section 8.

9.1. Overview of the visibility technique

The visibility technique allows invariants for mutually dependent peers, by weakening the restrictions on admissible invariants. In addition, it removes the restriction on what fields a method can assign to. For example, the visibility technique allows the invariant of a `Person` object to depend on the state of its spouse (as in Fig. 14). Furthermore, methods, such as `marry`, are allowed to directly assign to a field of the new spouse, `p`.

For soundness, the visibility technique must impose more and stronger proof obligations. For example, allowing the invariant of `Person` to depend on the `spouse` field of its spouse means that changing the state of a `Person` object could break the invariant of its spouse. Furthermore, allowing the `marry` method to directly assign to the `spouse` field of the new spouse, `p`, potentially breaks the invariant of the receiver object `p`, and any old spouses. The technique compensates, however, by requiring that a method making such a change re-establish the invariant of these objects. That is, the visibility technique imposes additional proof obligations on methods that ensure that these potentially broken invariants are preserved [3,14].

Moreover, the visibility technique allows one to determine what these additional proof obligations are modularly. The idea is to only add proof obligations that follow the program's module structure. So, an invariant I of an object X may depend on a field f of an object Y in X's context, provided that the declaration of I is visible where f is declared; this ensures that X's invariant I is visible in each method m that can assign to $Y.f$; such methods must preserve I for all relevant objects (which includes X). Most of these proof obligations can be discharged easily, since all invariants that do not depend on state that was modified by a method are trivially preserved.

For example, the invariant of a `Person` object (Fig. 14) can depend on the `spouse` field of its spouse. The additional proof obligations require one to show that each method m of class `Person` preserves the invariant of all `Person` objects in the context in which m is executed. In particular, one has to prove that `marry` preserves the invariants of the objects it modifies directly: `this` and `p`. Moreover, one has to prove that the invariants of all other objects in the same context are preserved, because these invariants are potentially affected by the modifications; in particular, any spouses of the modified objects might be affected. However, the precondition of the `marry` method avoids this complication by requiring that both objects have a null `spouse` field.

In summary, the key idea is that if an invariant declaration is visible in every method that might violate that invariant for a relevant object, then the obligations to preserve these invariants can be shown modularly.

To describe this idea more precisely, we need to describe a module system that defines an appropriate notion of visibility. We assume a simple module system, in which modules correspond to compilation units in Java. A declaration in class T *is visible in a method* m if T's module is imported by the module that contains m.[4]

Given this notion of visibility, we can summarize the main idea behind the visibility technique as follows.

Definition 9.1 (*Visibility Principle*). An invariant I *obeys the visibility principle* if I's declaration is visible in every method whose executions might violate I for a relevant object.

The visibility technique[5] combines the visibility principle and the ownership technique. This gives specifiers the flexibility to make trade-offs between expressiveness and the required proof obligations. We discuss these trade-offs in Section 10.

9.2. Encapsulation in the visibility technique

The visibility technique uses the concepts from the ownership model, but does not need the requirement that a method can only assign to fields of its receiver. In the visibility technique, a method is allowed to assign to fields of all objects in the context in which it is executed. This change from ownership encapsulation (Definition 6.1) is highlighted in bold below.

Definition 9.2 (*Visibility Encapsulated*). A field location $Y.f$ or an array location $Y[i]$ is *visibility encapsulated in an object* X if:

(1) $X = Y$ and the only assignments to the field location $X.f$ occur in method executions **in X's context**, or
(2) X owns Y.

For example, method `marry` (Fig. 14) is allowed to assign to `p.spouse`, as `p` is a `peer` object (since that is the default ownership annotation in the Universe type system), and hence `p` is in the same context as the receiver.

9.3. Visibility admissible invariants

The notion of visibility encapsulation permits methods to modify fields of other objects. As noted above, the invariants for such programs need to depend on mutable fields of non-rep objects in the same context. That is, the visibility admissible access expressions that can appear in a class C are characterized by the following regular expression: $(f_C \mid q_C\, q^* g \mid r_C(r \mid p)^* g)\, c^*$, where f is a field, q denotes `peer` field or array accesses, g is a non-constant field or array access, r and p are field or array accesses declared with a `rep` or `peer` annotation, respectively, c denotes constant field accesses, and the subscript C indicates that the field is declared in class C.

Definition 9.3 (*Visibility Admissible*). Let $E = g_0 \ldots g_N$ be an access expression occurring in a class C and M, $0 \leq M \leq N$, be the minimal index such that for each i, $M < i \leq N$, g_i is a constant field access. Then E is *visibility admissible* if the field name g_0 is declared in C and $g_0 \ldots g_M$ matches one of the following cases:

(1) $g_i, 0 \leq i < M$ **are field accesses of a `peer` type**, or
(2) $M > 0$, the pivot field g_0 is of a `rep` type, and all fields and array components in $g_1, \ldots, g_{M-1}$ are of a `rep` or `peer` type.

An invariant declaration in class C is *visibility admissible* if each of its access expressions is **visibility admissible, and if, for each prefix of an access expression that appears in the invariant and matches form (1), the invariant is visible in the class that declares the field g_M.**

[4] We assume that each module imports itself. Visibility must not be confused with accessibility: Each accessible declaration is visible, but declarations declared to be private will be visible but not accessible.

[5] The name "visibility technique" comes from the visibility principle, not from visible states. There are also verification techniques that are based on the visibility principle, but do not use a visible state semantics [13,39].

Access expressions of form (1) use the visibility principle to generalize form (1) of the definition of ownership admissible access expressions (Definition 6.2). This is a strict generalization, as can be seen by setting $M = 0$. This generalization allows the invariant of an object to depend on locations of other objects in the same context, which is the reason why form (1) uses `peer` accesses. Access expressions of form (1) are statically distinguishable from those of form (2), since only form (1) can consist of a single field name, and when both have length $M > 0$, then the former must start with a `peer` field name, while the latter must start with a `rep` field name. Access expression form 2 is identical to definition of ownership admissible field accesses (Definition 6.2) with the extension to readonly references (see Section 7.2).

The invariant in class `Person` (Fig. 14) is visibility admissible, because `spouse.spouse` is a visibility admissible access expression of form (1), and because `Person`'s invariant is visible in the class that declares `spouse`, namely `Person`.

An invariant that mentions `spouse.purse.amount` in class `Person` would not be visibility admissible because this access expression matches neither form (2) (`spouse` is not `rep`) nor form (1) (`amount` is non-constant and `purse` is not `peer`).

The invariant of class `ListIterator` (Fig. 11) is visibility admissible, because `theList.array` and `theList.n` are visibility admissible access expressions of form (1), assuming that `ListIterator` is visible in class `List`, where `array` and `n` are declared.

9.4. Semantics

The visibility technique uses the relevant invariant semantics (Definition 6.4).

9.5. Visibility proof technique

The visibility proof technique describes how to prove correctness of visibility admissible invariants with respect to the relevant invariant semantics. The ownership proof technique (Definition 6.5) is not sound for use with all visibility admissible invariants, as it relies on ownership encapsulation. The additional proof obligations needed are highlighted in bold in the following definition.

Definition 9.4 (*Visibility Proof Technique*). Let $inv(T)$ denote the conjunction of the invariants declared in a class T and its superclasses. Let $pre(o.p(\vec{e}))$ and $post(o.p(\vec{e}))$ denote the pre- and postcondition of a call to method p, determined by the static types of o and $\vec{e}$.

For each method m, and **for all classes T visible in m**, one must show that:

(O1) $inv(T)$ **holds for each T object in m's context** in each poststate of m's execution and m's postcondition holds in each poststate, and

(O2) for every call of a method $o.p(\vec{e})$ that appears in the body of m, $pre(o.p(\vec{e}))$ holds in the call's prestate, and if o is in m's context, then $inv(T)$ **also holds for each T object in m's context** in the prestate of the call to p. Moreover,

(O3) for each method m, and for each method call $o.p(\vec{e})$, appearing in m, if o has a readonly type, then the object referenced by o is inside the context that contains m's `this` object.

For the proof of each method m, one may assume that:

(A1) the invariants of all allocated objects that are relevant to the execution of m and m's precondition hold in m's prestate, and

(A2) after each call of a method $o.p(\vec{e})$, $post(o.p(\vec{e}))$ holds and all allocated objects that are relevant to the execution of $o.p(\vec{e})$ satisfy their invariants.

These proof obligations include the proof obligations of the ownership technique (Definition 6.5 and its extension for readonly references). To see this, let m be declared in class C. Then C and C's superclasses are visible in m. Moreover, the current receiver object of m is a C object in the context in which m is executed. So $inv(C)$ must be established before calls in the body of m (O2) and $inv(C)$ must also be preserved by m (O1).

As an example of this proof technique, consider proving that (O1) `Person`'s `marry` method (Fig. 14) preserves the invariant of all relevant `Person` objects, including the argument `p`. The proof starts by assuming (A1) that `Person`'s

invariant holds for both `this` and `p`, and that the precondition of `marry` also holds. The precondition says that `p` and `this` are distinct, and that neither has a spouse. The assignment of `p` to the spouse field of `this` breaks the invariant of `this`, although since `p` is distinct, its invariant is not broken. Then the assignment of `this` to `p.spouse` re-establishes the invariant of `this` and preserves the invariant of `p`. No other `Person` objects have their invariants broken by this assignment, since they cannot depend on either `p` or `this`, by the precondition. This establishes (O1).

The soundness theorem for the visibility technique is similar to the ownership soundness theorem (Theorem 6.6), but deals with methods that assign to fields of objects other than the current receiver object and the larger set of admissible invariants. The proof of the theorem is contained in Appendix C.

Theorem 9.5 (*Visibility Invariant Soundness*). *Let **P** be a program that is type correct in the Universe type system. Suppose all invariants of **P** are visibility admissible and methods of **P** only assign to fields of* `this`*, array elements, and fields of peer objects.*

*If all invariants of **P** have been proved using the visibility proof technique, and if X is an object relevant to the execution of a method m, then X satisfies its invariant in the visible states of m's execution.*

According to this soundness theorem, the proof obligations of the visibility proof technique (Definition 9.4) suffice to guarantee the relevant invariant semantics (Definition 6.4), even in the presence of the more general visibility admissible invariants (Definition 9.3). In particular, it suffices to prove that a method *m* preserves the visible invariants of the objects in its context. By the definition of visibility admissible invariants, *m* cannot violate non-visible invariants of these objects.

10. Discussion

Two fundamental principles, which we discuss below, enable the modular verification of invariants.

Techniques based on the following encapsulation principle gain their modularity from protecting the dependees of an invariant from certain modifications that potentially break the invariant. Both the classical technique and the ownership technique are *encapsulation-based*.

Definition 10.1 (*Encapsulation Principle*). The invariant of object *X obeys the encapsulation principle* if it only depends on locations encapsulated in *X* and on locations for constant fields.

In contrast, techniques based on the visibility principle (Definition 9.1) gain their modularity from enforcing that all invariants that are potentially affected by a field update are visible in the method that contains the update. Therefore, it is possible to show modularly that these invariants are preserved.

The visibility technique presented in this paper uses both the visibility and the encapsulation principle. A *purely visibility-based* technique would allow only access expressions of form (1) of Definition 9.3. Our visibility technique also supports access expressions of form (2), which use ownership, that is, encapsulation.

In this section, we summarize the trade-offs between the two principles and discuss combinations of both.

10.1. Trade-offs

The choice between encapsulation-based and purely visibility-based techniques is essentially a trade-off between expressiveness and verification effort.

Encapsulation-based techniques support layered object structures in which the invariant of an object *X* depends only on locations of objects in deeper layers that are exclusively owned by *X*. Such invariants occur in many practical examples, in particular, aggregate objects. However, they cannot express properties of common implementations such as mutually recursive classes. In contrast, purely visibility-based invariants relate objects in the same layer, but require that the invariant is visible in all classes that declare mutable fields that the invariant depends on. Therefore, purely visibility-based invariants of client programs must not depend on array elements or fields of library classes, because the invariant is not visible in these classes (array elements behave like public fields of class `Object`).

The restrictions of encapsulation-based techniques lead to simple proof obligations limited to the invariant of the current receiver object (see Definitions 3.4 and 6.5). Visibility-based techniques generally impose proof obligations for all visible invariants, which is often tedious.

In both approaches, the proof obligations can be discharged in a modular fashion. That is, the proof obligations for a method m can be proved by using m's implementation and the invariants that are visible in m. Invariants that are not visible in m cannot be violated by executions of m.

If applied in isolation, both encapsulation-based and purely visibility-based invariants are too restrictive for many common implementations. Therefore, our visibility technique combines both approaches. A verification tool can inspect invariants syntactically and determine whether the simple proof obligations of the ownership proof technique (Definition 6.5) suffice or whether all proof obligations of the visibility proof technique (Definition 9.4) are required.

The visibility technique presented in Section 9 is one way of combining the encapsulation and visibility principles to an expressive and practical verification technique. Other combinations make different trade-offs between expressiveness and the number of proof obligations. For instance, the number of proof obligations can be reduced by restricting visibility-admissible invariants. Assume that form (1) of Definition 9.3 requires that all fields g_i are declared in classes in C's package and have private or default access. Then the only methods that potentially violate the C invariant of an object in the context in which the method executes, are methods declared in classes in C's package. Methods of classes not in C's package cannot assign to any of these fields, g_i. Consequently, it is sufficient to prove that a method m of a class T preserves the invariants declared in the classes in the package that contains T (instead of all visible invariants) for objects in the context in which m executes.

10.2. Example

The producer–consumer example in Figs. 15 and 16 illustrates the expressiveness of combining encapsulation-based and visibility-based invariants. It is adopted from Barnett and Naumann [39], who present a number of interesting implementations that can be handled by visibility techniques.

The classes `Producer` and `Consumer` are implemented in the same package, which allows them to mutually access their default access fields. A `Producer` and a `Consumer` object share a common ring buffer of natural numbers, implemented by an array. The producer stores the index `f` of the next free array element, and the consumer stores the index `n` of the array element that is consumed next. That is, array elements `n` to `f` − 1 contain products, whereas elements `f` to `n` − 1 are empty. The synchronization of producer and consumer is expressed by specifications. To allow specifications of the producer to refer to the index `n` of the consumer, producers store a reference to the associated consumer, and vice versa.

`Producer`'s invariant expresses that the elements in the buffer array are natural numbers. This invariant requires encapsulation of the array, which is done using the ownership technique.

`Consumer`'s invariant specifies that a producer and the associated consumer are correctly linked to each other (analogously to the `spouse` invariant, see Fig. 14) and that both refer to the same buffer. Since `Producer` and `Consumer` objects do not own each other, this invariant requires visibility, that is, `Consumer` has to be visible in class `Producer`. This requirement is met because the classes import each other.

Besides illustrating the visibility technique, this example is interesting because it demonstrates that the ownership model does not necessarily prevent different objects from sharing common data: although the buffer is owned by the producer, the consumer can directly access the buffer via a readonly reference.

11. Future work

An important topic for future work is the extension of the approach to invariants that can depend on model fields [14,22,40]. Model fields are specification-only fields. They are not part of the implementation. They allow one to specify properties of data structures without referring to their implementation. Müller's work [3] shows that the presented approach carries over to visibility-based invariants with model fields. Adapting the technique to ownership invariants over model fields is planned for future work, which might simplify the proof obligations in many practical cases.

Another topic is how to support static fields and global data, which requires a generalization of the ownership model used in this paper. We also plan to extend the ownership type system and our verification technique to enable ownership transfer.

To avoid problems with re-entrant method executions [28], the visibility technique imposes a proof obligation (O2 in Definition 9.4). This proof obligation requires that before a method m makes a call $o.p(\vec{e})$, if o is a peer object, then

```
import Consumer;
class Producer {
  /*@ spec_public rep @*/ int[] buf = new /*@ rep @*/ int[10];
  /*@ spec_public @*/ int f = 0;
  /*@ spec_public peer @*/ Consumer con = null;

  /*@ public invariant buf != null
    @    && (\forall int i; 0 <= i && i < buf.length; buf[i]>=0)
    @    && 0 <= f && f < buf.length
    @    && con == null ==> f == 0 ;
    @*/

  /*@ requires   con != null
    @         && con.n != (f+1) % buf.length
    @         && x >= 0 ;
    @ assignable buf[f], f;
    @ ensures    f == \old((f+1) % buf.length);
    @*/
  void produce(int x) {
    buf[f] = x;
    f = (f+1) % buf.length;
  }
}
```

Fig. 15. Implementation of the producer. The invariant constrains the values stored in the buffer.

m must re-establish the visible invariants of objects in m's context. This is less onerous than the classical technique, since it does not require re-establishing m's receiver object's invariant (and the invariants of other objects in m's context) when manipulating the receiver's owned representation objects. It remains to be seen how difficult the proof obligation that remains is in practice and whether some more sophisticated treatment of re-entrant method calls is needed.

History constraints [5], which are also found in JML, suffer from the same problems as the classical invariant technique when applied to object structures. We plan to extend the techniques presented here to history constraints and possibly also to more general temporal constraints.

An open problem that is left for future work is that the classical technique as well as our ownership and visibility techniques require that invariants cannot depend on inherited fields. That is, the invariant in a class C cannot depend on a field `this.`f, if f is declared in a superclass of C. The reason for this restriction is that superclass methods could assign to f, thereby breaking such an invariant, but C's invariant is generally not available when reasoning about the superclass.

There are two kinds of subclasses that typically would contain such invariants. One is a subclass that restricts the set of possible states of objects. For instance, one could have a subclass of class `List`, `NonEmptyList`, that requires the list to have at least one element. Its invariant stating this restriction would depend on the inherited field `n`. Consider adding a `remove` method to `List`. A call to such a method could break this invariant. The other kind of subclass relates inherited to newly declared fields. For instance a subclass of class `BagWithMax`, `BagWithHistoricMax`, could store the largest value that has ever been seen by the bag. Its invariant stating this property would say that the historic maximum is at least as large as `maxElem`. A call to `BagWithMax`'s `insert` method that assigns to `maxElem` could break this invariant.

One solution that has been proposed is to force the programmer to override the methods of the superclass that might potentially break the invariant [26]. We could adopt this approach, but not all superclass methods can be overridden in Java (e.g., final methods). So this approach works for some but not all examples. Moreover a soundness proof is still in progress [41].

Leino and his colleagues proposed type-indexed invariants [12]. With such invariants, one can specify that a method of class C preserves the invariants declared in C and its superclasses, but might break the invariants declared in

```
import Producer;
class Consumer {
  /*@ spec_public readonly @*/ int[] buf;
  /*@ spec_public @*/ int n;
  /*@ spec_public peer @*/ Producer pro;

  /*@ public invariant buf != null
    @           && 0 <= n && n < buf.length
    @           && pro != null && pro.con == this
    @           && pro.buf == buf;
    @*/

  //@ requires p != null && p.con == null;
  Consumer(/*@ peer @*/ Producer p) {
    buf = p.buf;
    pro = p;
    n = 0;
    pro.con = this;
  }

  /*@ requires   n != pro.f;
    @ assignable n;
    @ ensures    n == \old((n+1) % buf.length);
    @*/
  int consume() {
    int oldn = n;
    n = (n+1) % buf.length;
    return buf[oldn];
  }
}
```

Fig. 16. Implementation of the consumer. A readonly reference allows the consumer to directly access the buffer owned by the producer. The visibility-based invariant relates a consumer to its producer.

subclasses of C. Since this approach allows methods to break certain relevant invariants, its application to a visible state semantics is not immediate.

12. Related work

The concept of abstraction layers has a long history, going back at least to the THE operating system [42]. However, the present paper is not about the concept of layering itself, but about techniques for specifying invariants that allow one to structure a system in layers.

In this section, we discuss papers from the large literature on invariants that are directly related to invariants for layered object structures. Since the Universe type system is not a contribution of this paper, the discussion of related work on ownership models focuses on how our ownership and visibility techniques can be combined with different models.

12.1. Invariants

Although the B-Method [43] is only object-based, it follows the classical invariant approach in the sense that it does not allow a machine (ADT) to impose additional invariants on the machines it builds on. This forces users to refine machines to maintain extra invariants, which is inconvenient and would not always be possible in object-oriented languages (see Section 11).

The proof obligations for the classical invariants are, for instance, used in Meyer's work [6,7] and the KeY tool [8]. They permit invariants over arbitrary object structures, but without the restrictions presented here; hence their

technique is not sound. For example, their work allows invariants over object structures like in `List` and `BagWithMax`, but uses the classical proof technique. We have argued in Section 4 why this is not sound.

Liskov, Wing, and Guttag [4,5] use a slight adaptation of the classical technique which requires the invariant of class T only to depend on private fields declared in T. Their technique enables a sound treatment of invariants for simple recursive object structures such as singly linked lists. However, our technique allows invariants to depend on fields of other types, which is necessary in layered structures, as illustrated in Fig. 2. Liskov and Wing do not explicitly discuss the soundness issues related to invariants that depend on inherited fields. We forbid such dependencies.

Huizing and Kuiper [28] present a proof system that supports invariants for arbitrary object structures. Invariants are analyzed syntactically to determine which methods of a program could violate which invariants. However, this whole-program analysis is not modular.

The work by Leino and Nelson [14,15] has visibility-based invariants for object structures. It uses dependency declarations to formulate restrictions that enable sound modular reasoning. This work is not based on the notion of ownership, which makes it difficult to formulate the requirements for admissible invariants. The soundness proof for static dependencies (form (1) of Definition 6.2) is very complicated and there is no proof for the more interesting case of dynamic dependencies (form (2) of Definition 6.2). Moreover, this work does not use a visible state semantics: any method can violate invariants as long as the violation is specified in the method's modifies clause. Visible states make explicit in which states invariants can be assumed to hold, which is useful in practical specification languages.

We have discussed the work of Barnett et al. [12] in Section 1.3. Leino and Müller [13] as well as Barnett and Naumann [39] extended the work by Barnett et al. [12] to visibility-based invariants. Barnett et al.'s work [12] cannot handle mutually recursive structures. Visibility-based invariants give more flexibility, and in particular they permit all the examples we discussed above. However, they still suffer from the above-mentioned disadvantages.

Barnett and Naumann [39] propose two ways to reduce the number of proof obligations for visibility-based invariants. First, update guards specify conditions under which a field update is guaranteed not to violate the invariant of an object. Second, sets of dependent objects can be used to determine efficiently all objects whose invariant is potentially broken by a field update. Both approaches can be combined with our visibility technique to simplify verification.

DeLine and Fähndrich [44] use typestates to express consistency of objects. Typestates can essentially express classical invariants (Definition 3.2) and also allow invariants to depend on inherited fields. Moreover, different invariants can be specified for the different typestates of an object. Aliasing is handled by a type system with linearity and the adoption and focus model [45], which provides a controlled way of creating aliases and accessing aliased objects, loosening the rigid uniqueness requirements imposed by linear type systems. Typestates can be checked automatically by a type checker, but the technique does not support invariants for object structures, which limits its application to comparably simple consistency criteria.

Separation logic [46] and related approaches [47,48] allow one to express that a predicate depends only on certain objects of a heap. It has been used to reason about classical invariants of modules with one instance [49]. Although Parkinson and Bierman [50] extend separation logic to an object-oriented language, they do not address how to prove invariants. Applying separation logic to reason about invariants for layered object structures is considered ongoing work. Perhaps the relevant invariant semantics that we propose will help separation logic address this problem.

12.2. Ownership models and readonly references

The ownership model supported by ownership types [33,34] is similar to the model used in this paper. It does not support readonly references, but allows objects to reference objects in ancestor contexts. Like outgoing readonly references, these references can be used to invoke methods on receivers whose invariant is temporarily violated. Our verification techniques can be adapted to this model by forbidding calls of non-helper methods on outgoing references. The ownership parameters of the ownership type system allow one to determine statically whether a reference is outgoing.

SafeJava [32,51] supports multiple ownership: An object X and all associated instances of inner classes can access the objects owned by X. Since a class and its inner classes are mutually visible, the adaptation of the visibility technique to this model is straightforward.

Aldrich's work [52] structures contexts into domains, which can be public. Objects outside a context Γ can access objects inside Γ through Γ's owner or any object in one of Γ's public domains. For instance, iterators in a public

domain of a collection are accessible for clients of the collection. They can be allowed to reference the representation of the collection stored in another domain. Since objects outside a context Γ can modify objects inside Γ without going through Γ's owner, the encapsulation principle is violated. Consequently, the ownership proof technique is not sound for this model. However, the visibility technique can be adapted by requiring that the invariant of Γ's owner is visible in the classes of all objects in Γ's public domains and, therefore, in all clients that call methods on objects in the public domains.

Leino and Müller [13] use an ownership model that is very similar to our model. Although modifications of an object have to be initiated by its owner, this work does not restrict aliasing. Instead of using a type system, Leino and Müller's methodology encodes ownership dynamically via ghost fields, invariants, and new statements. Therefore, it can handle programming patterns that cannot be statically typed in the Universe type system. For instance, it supports ownership transfer, which allows additional examples such as merging two doubly-linked lists.

Leino and Müller achieve the equivalent to subclass separation by using pairs of an object X and a "typeframe" of X as owner of an object. For instance, if X is an object of a class C, then an object can be owned by the pair $[X, D]$, where D is a superclass of C. The same ownership information is necessary in our approach to generate proof obligations that guarantee that downcasts from readonly to rep types preserve subclass separation.

Soundness of our verification techniques does not require that pure methods are completely side-effect free. JML allows pure methods to create and initialize new objects. In Skoglund's readonly system [38], a pure method must not modify the state of objects reachable from its receiver object, but is allowed to modify other parameter objects. This requirement is weaker than JML's purity, but strong enough to guarantee soundness of our techniques.

Birka and Ernst [37] present a type system for reference immutability, which is very similar to our readonly references, but not integrated with an ownership type system. Birka and Ernst's type system allows pure methods to modify fields explicitly declared as `mutable`, for instance, to update local caches. Soundness of our ownership and visibility techniques can be achieved for this notion of purity by preventing invariants from referring to `mutable` fields.

Naumann [53] generalizes purity to observational purity, which allows pure methods to make modifications as long as these modifications cannot be observed by client code. His work can be combined with our verification techniques by considering dependencies of client invariants as observations. In particular, this means that an observationally pure method must not assign to `spec_public` fields that are potentially mentioned in client invariants.

The above discussion shows that our verification techniques are general enough to be adapted to new developments in ownership models, readonly references, and purity.

13. Conclusions

In this paper, we have generalized classical object invariants to invariants over layered object structures. The classical technique cannot handle even simple object structures that use layers of abstractions such as bags built on top of lists. To achieve this generalization, we use an ownership model for alias control. The ownership model directly prevents representation exposure. It is also used to define the relevant invariant semantics, which frees methods of lower layers from the obligation to preserve the invariants of higher layers, which enables modular verification.

Modular verification is significantly simplified by using an ownership model which makes the abstraction layers of the design be reflected in the structure of the heap. That is, ownership information documents design decisions about abstraction layers, similar to the way invariants document the design of data structures. By using an ownership type system, we force programmers to record their design in the program in a way that can be used for reasoning.

The essence of the technique presented in this paper is the set of guidelines for programming and code reviewing in Fig. 17. Using an ownership model like the Universe type system would automatically enforce Rule 1 of the guidelines. Rule 2 reflects the definition of visibility-based invariants (Definition 9.3). Rule 3 covers two cases. For static dependencies, it subsumes the classical technique. For dynamic dependencies it requires invariants to obey the visibility principle (Definition 9.1) so that modular checking remains possible. Otherwise, this rule would require a whole-program analysis for public fields.

Rule 4 of the guidelines corresponds to ownership encapsulation (Definition 6.1). If g is part of the representation of X, then the Universe type system guarantees requirements (a) and (b). Requirement (c) is one of the proof obligations for the relevant invariant semantics (Definition 6.5). Without such an alias controlling type system, the programmer

(1) Organize your system in layers. A method execution on an object should only modify objects in the same or underlying layers.
(2) The invariant of an object X should only depend on the states of objects in the layer that contains X or underlying layers.
(3) If the invariant of an object X depends on a field f of an object in the same layer (including X itself), determine all methods that can assign to f and make sure that these methods preserve the invariant of X.
(4) Consider an invariant declared in class C and let X be a C object in layer L. Suppose this invariant of X depends on a field or an array element g in a layer underlying L. Make sure that every method execution on a receiver in layer L that modifies g:
   (a) is executed on receiver X,
   (b) is declared in C, and
   (c) re-establishes the invariant before it terminates or calls a method on an object in L (including X itself).

Fig. 17. Informal guidelines summarizing our technique for handling invariants.

has to manually avoid representation exposure (external aliases into the representation of X). Often, this can be done by cloning objects that are exchanged between clients in the internal representation of X.

The relevant invariant semantics and the corresponding proof principle of dividing responsibility for preserving invariants between a method and its callers is fundamental. It enables modular verification of invariants of object structures.

The relevant invariant semantics can be applied to both ownership and visibility-based invariants. Both kinds of invariants, and even classical invariants can co-exist in the same specification language. Visibility-based invariants would be used when the representation of an object cannot be completely encapsulated. In all other cases, ownership or classical invariants would be used, and would lead to simpler proof obligations. A tool could determine syntactically which proof obligations are necessary without additional annotations in the specification language.

Acknowledgments

We thank Ádám Darvas, Werner Dietl, Sophia Drossopoulou, Doug Lea, and David Naumann, the members of IFIP Working Group 2.3, and the anonymous referees of this journal and two conferences for helpful comments on earlier versions of this paper. Leavens' work was supported by the National Science Foundation under grants CCR-0097907, CCR-0113181, CCF-0428078, and CCF-0429567.

Appendix A. Soundness proof for the classical technique

The following is the proof of the soundness theorem for classical invariants (Theorem 3.5).

Proof. Modular soundness can be proved by induction over the sequence of visible states of a program execution. The base case is the sequence of length one consisting of an initial state. In initial states there are no allocated objects, and thus the result holds.

For the inductive case, the induction hypothesis is that all allocated objects satisfy their invariants in executions of length k ($k \leq n$). Without loss of generality, let us suppose, that there is a method m such that the nth visible state S_n is either m's prestate or the poststate of a method called in m. We have to show that all allocated objects satisfy their invariants in visible state S_{n+1}. Let m be declared in class C. Recall that $inv(C)$ is C's invariant. Visible state S_{n+1} is the prestate of a method called in m or the poststate of m. From the classical proof technique, we know that $inv(C)$ holds in state S_{n+1} for the current receiver object X. It remains to be shown that: (1) the other conjuncts of X's invariant (if X is an instance of a proper subclass of C) and (2) the invariants of all other allocated objects hold in visible state S_{n+1}.

According to classical encapsulation, the only locations that m can assign to between the visible states S_n and S_{n+1} are either array locations or non-constant fields f of X where f is declared in C or its superclasses (the static type of the receiver object in m is C). Classical invariants cannot depend on array locations. An assignment to $X.f$ with f declared in C or its superclasses cannot violate conjuncts of X's invariant that are declared in subclasses of C, because classical subclass invariants must not depend on superclass fields such as f. Thus, these conjuncts remain

valid, showing (1). Invariants of objects Y that are different from X cannot be violated by assignments to $X.f$, because such invariants may only depend on locations of Y. Thus, such invariants remain valid, showing (2). □

Appendix B. Soundness proof for the ownership technique

The following is the proof of the soundness theorem for ownership invariants (Theorem 6.6). The proof is analogous to the soundness proof for classical invariants given above.

Proof. We say that an execution state S *has context* Γ if S is the pre- or poststate of a method m executing in Γ, and that an object *is relevant in* S if it is relevant to the execution of m.

The key difference from the inductive proof for classical invariants is that different objects can be relevant in two different execution states, depending on whether these states are pre- or poststates of methods executed in the same or in different contexts. This leads to the following adapted induction hypothesis: For executions of length k ($k \leq n$), all allocated objects *that are relevant in state* S_k satisfy their invariants. We have to show that all allocated *relevant* objects satisfy their invariants in visible state S_{n+1}.

Without loss of generality, we can assume that visible state S_{n+1} is either the poststate of a method m executed in a context Γ on a receiver X, or the prestate of a method called in m on a receiver X. In the latter case, by Corollary 5.2, visible state S_{n+1} can either have context Γ or the context owned by X, Γ_X. Let m be declared in a class C, and let $inv(C)$ be defined as in Definition 6.5. We have to consider two cases for the induction step.

Case 1: The visible state S_{n+1} has context Γ. For the induction step, we consider the latest predecessor state S_j of state S_{n+1} that has context Γ. This state is either the prestate of m's execution or the poststate of a method called by m executing in Γ. Since states S_j and S_{n+1} have the same context, Γ, all objects that are relevant in state S_{n+1} are also relevant in state S_j, which implies that their invariants hold in state S_j.[6] By the proof obligations, we know that $inv(C)$ holds in state S_{n+1} for the current receiver object X. It remains to be shown that: (1) the other conjuncts of X's invariant hold (if X is an instance of a proper subclass of C) and (2) the invariant of all other allocated relevant objects hold in state S_{n+1}.

According to context encapsulation, between the visible states S_j and S_{n+1} method m can:

(a) assign to locations denoted by fields of X,
(b) assign to locations of arrays in context Γ,
(c) assign to locations of arrays in Γ_X, and
(d) modify locations inside Γ_X by executing methods in Γ_X.

Cases (a) and (b) are analogous to the classical soundness proof.

For cases (c) and (d), we prove that invariants of objects inside Γ_X, invariants of objects inside child contexts of Γ other than Γ_X, and invariants of objects in Γ (but not in child contexts) are preserved.

Objects inside Γ_X. In case (c), by Corollary 5.1 and Definition 6.2 the invariants of objects inside Γ_X cannot depend on an array element in Γ_X. Therefore, there are no invariants of objects inside Γ_X that could be violated by such an assignment.

In case (d), the induction hypothesis implies that the invariants of objects inside Γ_X hold in state S_n. The statements executed between S_n and S_{n+1} are part of m's body. As we do not allow X to directly modify objects inside Γ_X, their invariants are still valid in S_{n+1}.

Objects inside child contexts of Γ *other than* Γ_X. In cases (c) and (d) only locations of objects inside Γ_X can be modified (for case (d), this follows from context locality, Corollary 5.1). According to the definition of admissible invariants (Definition 6.2), invariants of objects in child contexts of Γ other than Γ_X must not depend on locations of objects inside Γ_X. Therefore, an invariant of an object in such a context must not depend on mutable locations of objects inside Γ_X and thus cannot be violated by modification of locations of objects inside Γ_X.

Objects in Γ. In cases (c) and (d), we prove (1) and (2) above by showing that the invariant declared in class C of object X is the only invariant of objects in Γ that could be affected by the modification of locations inside the context

[6] Since we are ignoring constructors in this paper, this argument is simplified, as it ignores object allocation. For the objects that are allocated in state $n+1$ but not in state j, we can assume that the constructors that initialized these objects established their invariants. Technically, constructor calls are handled analogously to method invocations [3].

Γ_X, which are the only locations that could be modified, as explained above. Assume there is an invariant declared in class D of an object Y in Γ that is affected, then we show $X = Y$ and $C = D$:

If this invariant depends on a mutable location inside Γ_X, the dependency must have a `rep` pivot p because the dependency is dynamic (form (2) of Definition 6.2). According to Definition 6.2, the field p is declared in the same class as the invariant, that is, in D. Since $Y.p$ references an object in Γ_X, Y is the owner of Γ_X (by the ingoing reference invariant Corollary 5.1). Consequently, we have $X = Y$ (as each context has at most one owner). Furthermore, according to subclass separation (Corollary 5.3), m can only modify a location reachable from $X.p$ if m is also declared in D. This implies $C = D$.

Case 2: The visible state S_{n+1} has a child context Γ_X of Γ. The nth state S_n is either the prestate of m or the poststate of a method called in m. That is, state S_n either has context Γ or Γ_X, and all objects that are relevant in state S_{n+1} (that is, that are inside Γ_X) are also relevant is state S_n. Therefore, all invariants relevant in state S_{n+1} hold in state S_n. According to Corollary 5.2 and the rule that a method can assign only to fields of `this`, the only locations that m can assign to between the visible states S_n and S_{n+1} are either fields f of m's receiver object X or locations of arrays in context Γ or Γ_X. Since X is in context Γ, it is not relevant in state S_{n+1}. Invariants of objects inside context Γ_X cannot depend on elements of arrays in Γ or Γ_X (Definition 6.2). Consequently, all allocated relevant objects satisfy their invariants in visible state S_{n+1}. □

Appendix C. Soundness proof for the visibility technique

The following is the proof of the soundness theorem for the visibility technique (Theorem 9.5). The proof is analogous to the soundness proof for ownership invariants given above. In particular, we prove the same induction hypothesis: For executions of length k ($k \leq n$), all allocated objects that are relevant in state S_k satisfy their invariants. We have to show that all allocated relevant objects satisfy their invariants in visible state S_{n+1}.

Proof. Without loss of generality, we can assume that visible state S_{n+1} is either the poststate of a method m executed in a context Γ on a receiver X, or the prestate of a method called in m on a receiver X.

If m is a pure method, the proof is trivial: we consider the latest predecessor state S_j of state S_{n+1} that has context Γ. This state is either the prestate of m's execution or the poststate of a method called by m executing in Γ. Since states S_j and S_{n+1} have the same context Γ, all objects that are relevant in state S_{n+1} are also relevant in state S_j, which implies that their invariants hold in state S_j. Since m does not modify any locations, the invariants still hold in state S_{n+1}.

If m is not pure, we know by Corollary 5.2 that visible state S_{n+1} has either context Γ or the context owned by X, Γ_X. We have to consider two cases for the induction step.

Case 1: The visible state S_{n+1} has context Γ. For the induction step, we consider the latest predecessor state S_j of state S_{n+1} that has context Γ. Again, we have that all objects that are relevant in state S_{n+1} satisfy their invariants in state S_j. From the proof obligations, we know that any T object Z in Γ satisfies $inv(T)$ in state S_{n+1}. It remains to be shown that: (1) the other conjuncts of Z's invariant hold (if Z is an instance of a proper subclass of T) and (2) the invariant of all objects in descendants of Γ hold in state S_{n+1}.

According to context encapsulation, between the visible states S_j and S_{n+1} method m can:

(a) assign to locations denoted by fields of objects in Γ,
(b) assign to locations of arrays in context Γ,
(c) assign to locations of arrays in Γ_X, and
(d) modify locations inside Γ_X by executing methods in Γ_X.

In case (a), assume that m assigns to a location $Y.f$. According to the definition of admissible invariants (Definition 9.3), the invariant declared in class D of an object Z can depend on $Y.f$ only if Z and Y are in the same context, Γ, and the invariant of D is visible in the class that declares f. Since m can assign to f, f is visible in m and, therefore, the invariant of D is also visible in m. Invariants in subclasses of D that are not visible in m cannot depend on $Y.f$ and are, thus, not affected by the update of $Y.f$, which implies (1). Invariants of objects in descendants of Γ cannot depend on $Y.f$, which implies (2).

Cases (b), (c), and (d) are analogous to the soundness proof for ownership invariants (Appendix B).

Case 2: The visible state S_{n+1} has a child context Γ_X of Γ. The nth state S_n is either the prestate of m or the poststate of a method called in m. That is, state S_n either has context Γ or Γ_X, and all objects that are relevant in state

S_{n+1} (that is, that are inside Γ_X) are also relevant in state S_n. Therefore, all invariants relevant in state S_{n+1} hold in state S_n. According to Corollary 5.2 and the rule that a method can assign only to fields of objects in the context in which it executes, the only locations that m can assign to between the visible states S_n and S_{n+1} are either fields f of objects Y in context Γ or locations of arrays in context Γ or Γ_X. Invariants of objects inside context Γ_X can neither depend on locations $Y.f$ of objects in Γ nor on elements of arrays in Γ or Γ_X (Definition 9.3). Consequently, all allocated relevant objects satisfy their invariants in visible state S_{n+1}. □

References

[1] C.A.R. Hoare, Proof of correctness of data representations, Acta Informatica 1 (1972) 271–281.
[2] M.D. Ernst, J. Cockrell, W.G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Transactions on Software Engineering 27 (2) (2001) 1–25.
[3] P. Müller, Modular Specification and Verification of Object-Oriented programs, in: Lecture Notes in Computer Science, vol. 2262, Springer-Verlag, 2002.
[4] B. Liskov, J. Guttag, Abstraction and Specification in Program Development, MIT Press, 1986.
[5] B. Liskov, J.M. Wing, A behavioral notion of subtyping, ACM Transactions on Programming Languages and Systems 16 (6) (1994) 1811–1841.
[6] B. Meyer, Eiffel: The Language, Prentice Hall, 1992.
[7] B. Meyer, Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.
[8] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P.H. Schmitt, The KeY tool, Software and System Modeling 4 (1) (2005) 32–54.
[9] J. Hogg, Islands: Aliasing protection in object-oriented languages, in: A. Paepcke (Ed.), Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 1991, pp. 271–285, SIGPLAN Notices 26 (11).
[10] J. Noble, J. Vitek, J.M. Potter, Flexible alias protection, in: E. Jul (Ed.), ECOOP '98: Object-Oriented Programming, in: Lecture Notes in Computer Science, vol. 1445, Springer-Verlag, 1998, pp. 158–185.
[11] B. Bruegge, A.H. Dutoit, Object-Oriented Software Engineering, 2nd edition, Prentice Hall, 2004.
[12] M. Barnett, R. DeLine, M. Fähndrich, K.R.M. Leino, W. Schulte, Verification of object-oriented programs with invariants, Journal of Object Technology (JOT) 3 (6). http://www.jot.fm.
[13] K.R.M. Leino, P. Müller, Object invariants in dynamic contexts, in: M. Odersky (Ed.), European Conference on Object-Oriented Programming (ECOOP), in: Lecture Notes in Computer Science, vol. 3086, Springer-Verlag, 2004, pp. 491–516.
[14] K.R.M. Leino, Toward reliable modular programs, Ph.D. Thesis, California Institute of Technology, 1995.
[15] K.R.M. Leino, G. Nelson, Data abstraction and information hiding, ACM Transactions on Programming Languages and Systems 24 (5) (2002) 491–553.
[16] W. Dietl, P. Müller, Exceptions in ownership type systems, in: E. Poll (Ed.), Formal Techniques for Java-like Programs, 2004, pp. 49–54.
[17] C. Boyapati, R. Lee, M. Rinard, Ownership types for safe programming: Preventing data races and deadlocks, in: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2002, pp. 211–230.
[18] C. Flanagan, S.N. Freund, Atomizer: A dynamic atomicity checker for multithreaded programs, in: ACM Symposium on Principles of Programming Languages and Systems, ACM Press, 2004, pp. 256–267. URL: http://doi.acm.org/10.1145/964001.964023.
[19] B. Jacobs, K.R.M. Leino, W. Schulte, Verification of multithreaded object-oriented programs with invariants, in: Specification and Verification of Component-Based Systems (SAVCBS), Technical Report 04-09, Department of Computer Science, Iowa State University, 2004, pp. 2–9. Available from http://www.cs.iastate.edu/~leavens/SAVCBS/2004/savcbs04.pdf.
[20] E. Rodríguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, Robby, Extending JML for modular specification and verification of multi-threaded programs, in: A.P. Black (Ed.), European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 3586, Springer-Verlag, 2005, pp. 551–576.
[21] W. Weihl, B. Liskov, Implementation of resilient, atomic data types, ACM Transactions on Programming Languages and Systems 7 (2) (1985) 244–269.
[22] G.T. Leavens, A.L. Baker, C. Ruby, JML: A notation for detailed design, in: H. Kilov, B. Rumpe, I. Simmonds (Eds.), Behavioral Specifications of Businesses and Systems, Kluwer Academic Publishers, 1999, pp. 175–188.
[23] G.T. Leavens, A.L. Baker, C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, Tech. Rep. 98-06-rev28, Department of Computer Science, Iowa State University, July 2005. See http://www.jmlspecs.org. URL: ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/TR.ps.gz.
[24] J.V. Guttag, J.J. Horning, Larch: Languages and Tools for Formal Specification, Springer-Verlag, 1993.
[25] A. Goldberg, D. Robson, Smalltalk-80, in: The Language and its Implementation, Addison-Wesley Publishing Co., Reading, Mass, 1983.
[26] C. Ruby, G.T. Leavens, Safely creating correct subclasses without seeing superclass code, in: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, ACM SIGPLAN Notices 35 (10) (2000) 208–228. URL: ftp://ftp.cs.iastate.edu/pub/techreports/TR00-05/TR.ps.gz.
[27] A. Poetzsch-Heffter, Specification and verification of object-oriented programs, Habilitation Thesis, Technical University of Munich, January 1997.
[28] K. Huizing, R. Kuiper, Verification of object-oriented programs using class invariants, in: E. Maibaum (Ed.), Fundamental Approaches to Software Engineering, in: Lecture Notes in Computer Science, vol. 1783, Springer-Verlag, 2000, pp. 208–221.
[29] D.L. Detlefs, K.R.M. Leino, G. Nelson, Wrestling with rep exposure, Research Report 156, Digital Systems Research Center, 1998.

[30] J. Aldrich, V. Kostadinov, C. Chambers, Alias annotations for program understanding, in: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), ACM Press, 2002, pp. 311–330.
[31] B. Bokowski, J. Vitek, Confined types, in: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), ACM Press, 1999, ACM SIGPLAN Notices 82–96.
[32] C. Boyapati, Safejava: A unified type system for safe programming, Ph.D. Thesis, MIT. Available from pmg.lcs.mit.edu/~chandra/publications/, 2004.
[33] D. Clarke, Object ownership and containment, Ph.D. Thesis, University of New South Wales, 2001.
[34] D.G. Clarke, J.M. Potter, J. Noble, Ownership types for flexible alias protection, in: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), ACM Press, 1998, ACM SIGPLAN Notices 33 (10) 48–64.
[35] P. Müller, A. Poetzsch-Heffter, Universes: A type system for alias and dependency control, Tech. Rep. 279, Fernuniversität Hagen, 2001.
[36] W. Dietl, P. Müller, Universes: Lightweight ownership for JML, Journal of Object Technology (JOT) 4 (8).
[37] A. Birka, M.D. Ernst, A practical type system and language for reference immutability, in: Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), ACM Press, 2004, pp. 35–49.
[38] M. Skoglund, Sharing objects by read-only references, in: H. Kirchner, C. Ringeissen (Eds.), Algebraic Methodology and Software Technology (AMAST), in: Lecture Notes in Computer Science, vol. 2422, Springer-Verlag, 2002, pp. 457–472.
[39] M. Barnett, D. Naumann, Friends need a bit more: Maintaining invariants over shared state, in: D. Kozen (Ed.), Mathematics of Program Construction, in: Lecture Notes in Computer Science, vol. 3125, Springer-Verlag, 2004, pp. 54–84.
[40] P. Müller, K.R.M. Leino, A verification methodology for model fields, in: P. Sestoft (Ed.), European Symposium on Programming, ESOP, in: Lecture Notes in Computer Science, vol. 3924, Springer-Verlag, 2006, pp. 115–130.
[41] C. Ruby, Safely creating correct subclasses without seeing superclass code, Ph.D. Thesis, Iowa State University, Ames, Iowa, 2006 (in press).
[42] E.W. Dijkstra, The structure of the THE multi-programming system, Communications of the ACM 9 (3).
[43] J.-R. Abrial, The B Book—Assigning Meanings to Programs, Cambridge University Press, 1996.
[44] R. DeLine, M. Fähndrich, Typestates for objects, in: M. Odersky (Ed.), European Conference on Object-Oriented Programming. ECOOP, in: Lecture Notes in Computer Science, vol. 3086, Springer-Verlag, 2004, pp. 465–490.
[45] M. Fähndrich, R. DeLine, Adoption and focus: practical linear types for imperative programming, in: Programming Language Design and Implementation (PLDI), ACM Press, 2002, pp. 13–24.
[46] J.C. Reynolds, Separation logic: A logic for shared mutable data structures, in: Proceedings Seventeenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, 2002, pp. 55–74.
[47] A. Ahmed, L. Jia, D. Walker, Reasoning about hierarchical storage, in: Logic In Computer Science (LICS), IEEE Computer Society Press, 2003, pp. 33–44.
[48] S.S. Ishtiaq, P.W. O'Hearn, BI as an assertion language for mutable data structures, in: Principles of Programming Languages (POPL), ACM Press, 2001, pp. 14–26.
[49] P.W. O'Hearn, H. Yang, J.C. Reynolds, Separation and information hiding, in: Principles of Programming Languages (POPL), ACM Press, 2004, pp. 268–280.
[50] M. Parkinson, G. Bierman, Separation logic and abstraction, in: Principles of Programming Languages (POPL), ACM, 2005, pp. 247–258.
[51] C. Boyapati, B. Liskov, L. Shrira, Ownership types for object encapsulation, in: Principles of Programming Languages (POPL), ACM Press, 2003, pp. 213–223.
[52] J. Aldrich, C. Chambers, Ownership domains: Separating aliasing policy from mechanism, in: M. Odersky (Ed.), European Conference on Object-Oriented Programming, ECOOP, in: Lecture Notes in Computer Science, vol. 3086, Springer-Verlag, 2004, pp. 1–25.
[53] D. Naumann, Observational purity and encapsulation, in: M. Cerioli (Ed.), Fundamental Aspects of Software Engineering (FASE), in: Lecture Notes in Computer Science, vol. 3442, Springer-Verlag, 2005, pp. 190–204.