Verification of Programs with Inspector Methods

Bart Jacobs and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{bart.jacobs,frank.piessens}@cs.kuleuven.be

Abstract. Most classes in an object-oriented program provide access to an object's state through methods, so that client code does not depend on and cannot interfere with the object's internal representation composed of fields and internal component objects. Methods used for this purpose are sometimes called *inspector methods*. In order to extend the benefits of inspector methods to specifications, the method contracts of non-inspector methods may be expressed using inspector methods, hence providing support for state abstraction in specifications.
In this paper, we propose an approach to the verification of programs that use inspector methods in method contracts and object invariants. Inspector methods may have parameters, and they may depend on the state of objects passed as arguments. Our approach builds on the Boogie methodology for object invariants and ownership.
Performing state abstraction in a programming language that allows aliasing through object references poses a framing problem. Specifically, client code needs to be able to tell whether modifying a given object or calling a given method may affect the value of a given inspector method call. We solve this by modeling inspector methods as functions that take as arguments only those parts of the heap on which they depend. Thanks to a novel logical encoding of the heap, we can do this without breaking information hiding, even in cases where inspector methods depend on internal component objects.
The core of our approach has been implemented in a custom build of the Spec# program verifier.

1 Introduction

Consider the program in Figure 1. Class *Cell* provides access to the state of a *Cell* object using method *getX*. It also uses *getX* to specify the effect of the class's constructor and of the *setX* method. This makes it possible to prove the correctness of the client program using a proof that does not depend on the internal representation of the *Cell* object's state using field x. As a result, when class *Cell*'s internal representation is changed, only class *Cell* needs to be reverified. The client program's proof remains valid.

In this paper, we concern ourselves with how to prove the correctness of programs such as the one in Figure 1.

Note that one of the key problems that needs to be solved by our verification approach is the *framing* problem. Specifically, in order to prove the assertion in

```
class Cell {
  int x;
  inspector int getX()
  { return x; }
  Cell(int value)
    ensures getX() = value;
  { x := value; }
  void setX(int value)
    modifies this.*;
    ensures getX() = value;
  { x := value; }
}
```

```
Cell c1 := new Cell(0);
int y := c1.getX();
assert y = 0;
Cell c2 := new Cell(5);
c1.setX(10);
assert c1.getX() = 10;
assert c2.getX() = 5;
```

Fig. 1. A class specified using an inspector method, and a client program

Figure 1 that $c2.getX() = 5$, it must be encoded in the verification logic that $c2.getX()$ does not depend on any objects modified by method call $c1.setX(10)$.

The remainder of the paper is structured as follows. We introduce our approach in five versions, each extending the previous one. In Section 2, we address the framing problem in its basic form (Version 1). In Section 3, we introduce the Boogie methodology [1] for object invariants (Version 2) and ownership, and we show how to solve the framing problem for inspector methods that may depend on owned objects (Version 3). In Section 4, we show how we allow inspector methods to depend on the state of objects passed as arguments (Version 4). In Section 5, we deal with a number of formal details. In Section 6, we describe our approach to inheritance (Version 5). In the final sections, we discuss related work and offer a conclusion.

2 Framing

In this section, we show how our approach addresses the framing problem in the simple setting where an inspector method may depend only on the fields of the receiver object. In later sections, we extend the approach to allow inspector methods to depend on fields of transitively owned objects and objects passed as arguments.

Our solution consists of three components: the way the heap is modeled, the way inspector method calls are modeled, and the way non-inspector method calls (which may modify the program state) are modeled.

In our verification logic, we represent the heap as a function that maps object references to *object states*. For the simple setting we discuss here, an object state is a function that maps field names to field values. Later sections extend the notion of object state to deal with transitively owned objects.

We treat inspector methods as follows. For each inspector method declared in the program, we introduce a function symbol in the verification logic, as well

as an axiom, derived from the inspector method's body, that defines the function symbol's meaning. All information on which the inspector method depends is passed as arguments to the corresponding function. Hence, in the simple setting we discuss here, we pass only the receiver's reference and state. For example, the assertion

$$\textbf{assert } c2.getX() = 5;$$

is modeled as

$$\textbf{assert } Cell\_getX(c2, Heap[c2]) = 5;$$

The final ingredient to our approach is the way non-inspector method calls are modeled in our verification logic. The post-state heap of a non-inspector method call is assumed to be an arbitrary new heap, constrained only by the method's declared postconditions and by an implicit postcondition called the *frame condition*. The frame condition says that for all objects o that were allocated in the pre-state and that are not listed by the method's modifies clause, the object's post-state is equal to its pre-state. Formally:

$$\forall o \bullet \textbf{old}(Alloc[o] \wedge o \notin W) \Rightarrow Heap[o] = \textbf{old}(Heap[o])$$

where W denotes the set of objects listed by the modifies clause. (We allow only items of the form $o.*$ in modifies clauses. Also, note that a constructor's modifies clause is always considered to implicitly include **this**.$*$.)

We can now prove the program of Figure 1. Let *Heap* and $Heap'$ denote the heap in the pre-state and the post-state, respectively, of the *setX* call. We need to prove $Cell\_getX(c2, Heap'[c2]) = 5$. From the postcondition of *Cell*'s constructor, we have $Cell\_getX(c2, Heap[c2]) = 5$. Finally, from the frame condition of the *setX* call, we have $Heap'[c2] = Heap[c2]$, which allows us to arrive at our goal by substitution.

3 Object invariants and ownership

In this section, we integrate inspector methods with the support for object invariants and ownership provided by the Boogie methodology [1]. The example in Figure 2 motivates and illustrates the approach. (Note: the expression $(a : b)$ denotes the range of integers i where $a \leq i < b$.)

An object of class *IntList* in Figure 2 represents a container for a list of integers. Internally, the integers are stored in an array *elems*. As is common, the array may be larger than the length of the list to minimize the number of heap allocations when adding or removing elements. The actual number of elements is stored in the *count* field.

3.1 Object Invariants

Methods that operate on an *IntList* object, such as the *add* method, need to know that *count* is never negative and never greater than the length of *elems*.

```
final class IntList {
  rep int[] elems;
  int count;

  invariant 0 ≤ count ∧ count ≤ elems.length;

  inspector int getCount() { return count; }
  inspector int getItem(int index)
    requires 0 ≤ index ∧ index < getCount();
  { return elems[index]; }

  derived_invariant 0 ≤ getCount();

  IntList(int[] xs)
    requires ¬xs.committed;
    ensures ¬this.committed ∧ this.inv;
    ensures getCount() = xs.length;
    ensures
      forall{int i in (0 : getCount()); getItem(i) = xs[i]};
  { ... }

  void add(int x)
    requires ¬this.committed ∧ this.inv;
    modifies this.∗;
    ensures ¬this.committed ∧ this.inv;
    ensures getCount() = old(getCount()) + 1;
    ensures
      forall{int i in old((0 : getCount())); getItem(i) = old(getItem(i))};
    ensures getItem(old(getCount())) = x;
  {
    unpack this;
    count++;
    ensureCapacity(count);
    elems[count − 1] := x;
    pack this;
  }

  ...
}

int[] xs := {1, 2, 3};
IntList list := new IntList(xs);
xs[0] := 5;
assert list.getItem(0) = 1;
```

Fig. 2. A class that illustrates object invariants, ownership, parameterized inspector methods, inspector method preconditions, and derived invariants. (Note: reference types are non-null types by default.)

It would be unfortunate to require the method's caller to guarantee this; this would cause the caller to depend on the internals of class *IntList*. To solve this problem, the Boogie methodology provides a mechanism called *object invariants* that allows a developer to expose conditions on internal state to clients in an *abstracted* form. Specifically, it allows a developer to declare an object invariant using the new **invariant** keyword, and in each object, it introduces a special boolean field *inv* and it restricts modifications of this field and the object's other fields in such a way that *inv* is only ever *true* at a time when the object invariant holds. Consequently, by exposing to clients only the *inv* field and requiring *inv* to be *true* on entry to a method, the method can rely on the internal object invariant without having to expose it. Note that methods are not allowed to assume without proof that object invariants hold on entry to the method; this would be unsound because of possible re-entrancy [1].

An object o for which $o.inv$ is *false* is called *mutable*; if $o.inv$ is *true*, the object is called *valid*. An assignment to a field $o.f$ is allowed only when o is mutable.

Field *inv* is initially *false*. It may be read only in method contracts, not in program code. Also, it may be updated only through the special new statements **pack** o; and **unpack** o;. These statements behave as follows (where $Inv(o)$ denotes o's object invariant):

pack o; $\equiv$	**unpack** o; $\equiv$
assert $\neg o.inv$;	**assert** $o.inv$;
assert $Inv(o)$;	$o.inv := false$;
$o.inv := true$;	

That is, the **pack** o; operation checks that o is mutable, and that o's object invariant holds. It then marks the object as valid. The **unpack** o; operation checks that o is valid. It then marks the object as mutable.

Provided that an object o's object invariant depends only on the fields of o, the semantics of **pack** and **unpack** together with the restriction on field assignments guarantee that whenever the object is valid (i.e. $o.inv$ is *true*), its object invariant holds. We call this property the soundness of the object invariant methodology.

Inspector methods and object invariants Non-inspector methods that rely on an object's invariant need to require the object's validity as a precondition. Whereas for non-inspector methods we provide the option of either requiring validity of a given object or not requiring it, for inspector methods we always require validity of the receiver object. That is, each inspector method implicitly gets a precondition saying that the receiver object is valid. We made this choice because supporting inspector methods that do not require the validity of their receivers would complicate the approach, and scenarios where the abstraction provided by inspector methods is required but the abstraction provided by object invariants is not, are probably rare.

Since it needs to be possible to evaluate an object invariant even in a state where it does not hold, we do not allow inspector method calls on **this** in an

object invariant. However, in addition to an object invariant, a class may declare a *derived invariant*, using one or more **derived_invariant** declarations. If a class declares a derived invariant, this implies a proof obligation that the derived invariant follows from the object invariant. Contrary to an object invariant, a derived invariant may include inspector method calls on **this**. If a class mentions private fields in its object invariant, it must declare the object invariant itself private, and as a result, other classes cannot use it in proofs. Still, the class can expose information to other classes in the form of public derived invariants, provided that the information is stated in terms of public inspector methods.

We could have allowed inspector method postconditions instead of, or in addition to, derived invariants. This would be equivalent in terms of expressiveness. However, since such postconditions would in general have to mention other inspector methods, to express relationships between inspector methods, it seems more natural to centralize this information at the class level.

Derived invariants (or an equivalent mechanism) serve to reduce specification effort. For example, if class *IntList* did not declare a derived invariant, each method that takes an *IntList* object *list* as an argument would have to specify $0 \leq list.getCount()$ in its precondition and postcondition.

3.2 Ownership

Does the client program provided in Figure 2 verify? Without an ownership system, the answer would be *yes*, regardless of whether *IntList*'s constructor copies *xs* into a new array or simply stores a reference to *xs* into the *elems* field. Clearly, this is unsound.

The cause of this unsoundness is the fact that the *getItem* inspector reads the elements of *elems*, even though, as discussed in Section 2, the function generated for the logical encoding of *getItem* takes only the state of **this**, not the state of the *elems* array, as a parameter.

In order to allow inspector methods like *getItem*, which depend on the state of objects other than the receiver object, we apply the Boogie methodology's *ownership system*, where an object can *own* other objects. Specifically, whenever an object o is valid, it *owns* the objects referred to by o's **rep** fields. For example, in Figure 2, whenever an *IntList* object o's *inv* field is **true**, o owns the array pointed to by $o.elems$.

We allow inspector methods to depend on their receiver object, as well as any objects directly or indirectly owned by it. However, in the verification logic, we still wish to pass just the state of the receiver object as an argument, as opposed to passing an additional argument for each owned object. The reason is that **rep** fields are typically private fields, so requiring clients to pass an additional argument for each **rep** field in inspector method function applications when verifying their code would break information hiding.

To make this work, we change the logical encoding of the heap. Object references still map to object states, but the notion of object state is extended to deal with ownership. An object o's object state is extended to contain a copy of the state of each of o's owned objects. Specifically, for each **rep** field $o.f$,

we introduce an additional field $o.f_{\text{state}}$ which maps to a copy of the state of the object referred to by $o.f$ whenever $o.inv$ is **true**. Of course these additional fields exist only in the logical encoding; they do not exist at run time. We do not change the program's run-time semantics.

This extended notion of object state allows inspector method results to be defined entirely in terms of the state of the receiver object. For example, the axiom that defines the function symbol for inspector method *getItem* is as follows:

$$\forall o, o_{\text{state}}, i \bullet \\ \quad \mathit{IntList\_getItem}(o, o_{\text{state}}, i) = o_{\text{state}}[\mathit{IntList\_elems}_{\text{state}}][i]$$

Clearly, using a copy of an object's state instead of the original is sound only if the copy is up-to-date whenever it is used. This is exactly what the heap consistency theorem below shows. The proof of this theorem relies on another aspect of the Boogie methodology. The methodology does not allow updates to fields of *committed* objects, i.e. objects owned by other objects.

An object p is committed if and only if there is some valid object o that has a **rep** field $o.f$ such that $o.f = p$. However, to simplify the verification logic, we track whether an object o is committed explicitly in the form of a boolean field $o.committed$. Like the *inv* field, this field can be read only in method contracts and cannot be assigned to explicitly in code.

To deal with ownership, we extend the meaning of the **pack** and **unpack** commands as follows:

pack o; $\equiv$
  assert $\neg o.committed \wedge \neg o.inv$;
  foreach (non-null **rep** field $o.f$)
    assert $\neg o.f.committed \wedge o.f.inv$;
  assert $Inv(o)$;
  foreach (non-null **rep** field $o.f$) {
    $o.f.committed := true$;
    $o.f_{\text{state}} := Heap[o.f]$;
  }
  $o.inv := true$;

unpack o; $\equiv$
  assert $\neg o.committed \wedge o.inv$;
  foreach (non-null **rep** field $o.f$)
    $o.f.committed := false$;
  $o.inv := false$;

Heap consistency As explained above, if the body of an inspector method dereferences an owned object **this**.f, we retrieve its state from a special field **this**.f_{state} instead of looking it up in the heap as usual. This is sound because **this** is *rep-consistent*:

Definition 1 (rep-Consistency). *An object o is* rep-consistent *if, for each non-null* **rep** *field $o.f$, it holds that*

$$o.f_{\text{state}} = Heap[o.f]$$

.

In fact, we have the following theorem:

Theorem 1 (rep-Consistency). *In each program state of each execution of each valid program, each valid object is rep-consistent.*

Proof. Since our approach is a conservative extension of the Boogie methodology [1], we may assume the known properties of the Boogie methodology. In particular, we know that objects pointed to by **rep** fields of valid objects are valid and committed.

We prove the theorem by induction over the length of an execution. This holds for the empty execution, since in the initial program state no object is valid. Now consider a non-empty execution. We now look at the final command performed in this execution:

- A field assignment $p.g := v;$. We know that p is mutable and rep-consistency involves only valid objects; therefore, this command does not invalidate the theorem.
- A **pack** p; operation. This operation establishes the rep-consistency of p, and it does not break the rep-consistency of the objects pointed to by the **rep** fields of p since changing $o.committed$ does not influence the rep-consistency of an object o.
- An **unpack** p; operation. Since p is made mutable, the theorem no longer applies to p. Also, since the Boogie methodology guarantees that committed objects have unique owners, changing the *committed* bits of p's owned objects does not invalidate the rep-consistency for any valid object $o \neq p$.
- No other commands modify existing objects.

Ownership and frame conditions Recall from Section 2 that each method gets an implicit postcondition, called the *frame condition*, that encodes in the verification logic that some objects are not modified by the method. The frame condition given in Section 2 was:

$$\forall o \bullet \mathbf{old}(Alloc[o] \wedge o \notin W) \Rightarrow Heap[o] = \mathbf{old}(Heap[o])$$

In the presence of ownership, a different frame condition is required. Specifically, we wish to allow a method that lists an object o in its modifies clause to modify not just o, but objects directly or indirectly owned by o as well. (We cannot require the method to list these owned objects in the modifies clause because of information hiding.) To achieve this, we follow the Boogie methodology [1] in allowing the method to modify any object that is *committed* in the pre-state, in addition to the objects listed in the modifies clause:

$$\forall o \bullet \mathbf{old}(Alloc[o] \wedge o \notin W \wedge \neg o.committed) \Rightarrow Heap[o] = \mathbf{old}(Heap[o])$$

4 Multi-dependent inspector methods

In this section, we show how the approach supports *multi-dependent* inspector methods, i.e. inspector methods that depend on the state of objects passed as arguments, in addition to the state of the receiver object.

A motivating example is shown in Figure 3. It shows part of the Microsoft .NET Framework's support for restricting the access partially trusted code has to system resources.[1] A *PermissionSet* object holds the *permissions* assigned to a given piece of code. Permissions are represented using objects that implement interface *Permission*. For example, a *SecurityPermission* object may represent permission to run, or permission to skip bytecode verification (or both, or neither).

Interface *Permission* declares an inspector method *isSubsetOf* that returns whether one permission of a given type is implied by another permission of the same type. Also, class *PermissionSet* declares an inspector method *contains* that returns whether the set contains a given permission.

Importantly, in the .NET Framework *Permission* objects are mutable. That is, a given *Permission* object may be made to represent different permissions at different points in time. However, a *PermissionSet* object contains specific permissions, not *Permission* objects. Therefore, the *contains* inspector method must be allowed to depend on the state of the *Permission* object passed as an argument, not just its identity.

For example, in the piece of client code shown at the bottom of Figure 3, permission to execute is added to a permission set, and the *contains* method returns *true* for the *Permission* object that represents this permission. However, if subsequently permission to skip verification is added to the *Permission* object, calling *contains* with the same object returns *false*. This shows the need for multi-dependent inspector methods.

The following issues arise when multi-dependent inspector methods are admitted: how are calls of such methods translated into the verification logic, how do we ensure consistency of the resulting logic, and how do we specify the abstract state of an object using multi-dependent inspector methods.

An inspector method is allowed to depend on the state of the receiver object, the objects that are passed as arguments for parameters marked **state**, and their transitively owned objects. A call of an inspector method is translated into an application of the corresponding function symbol with the following arguments:

- For each argument a to the inspector method call, there is an argument to the function symbol application that models the value a.
- In addition, for each argument a to the inspector method call for a parameter marked **state**, there is an argument to the function symbol application that models the state of the object a.

An inspector method implicitly gets a precondition that says that each argument a for a **state** parameter must be valid (i.e., $a.inv$ is *true*).

[1] We changed names to conform to Java naming conventions.

```
interface Permission {
  inspector boolean isSubsetOf(state Permission other)
    requires other.getClass() = getClass();
}
final class SecurityPermission implements Permission {
  static final int EXECUTION := 1;
  static final int SKIP_VERIFICATION := 2;

  int flags;
  inspector int getFlags() { return flags; }
  inspector boolean isSubsetOf(state Permission other) {
    return (flags & ~((SecurityPermission)other).flags) = 0;
  }
  derived_invariant
    forall{state SecurityPermission p;
      isSubsetOf(p) = ((getFlags() & ~p.getFlags()) = 0)};

  SecurityPermission(int flags)
    ensures getFlags() = flags;
  { this.flags := flags; }

  void setFlags(int flags)
    ensures getFlags() = flags;
  { this.flags := flags; }
}
final class PermissionSet {
  import Permission;
  ...
  inspector boolean contains(state Permission p) { ... }

  PermissionSet()
    ensures forall{state Permission p; ¬contains(p)};
  { ... }

  void setPermission(Permission p)
    ensures forall{state Permission q;
      contains(q) = (q.getClass() = p.getClass() ? q.isSubsetOf(p) : old(contains(q)))};
  { ... }
}

PermissionSet s := new PermissionSet();
Permission p := new SecurityPermission(SecurityPermission.EXECUTION);
s.setPermission(p);
assert s.contains(p);
p.setFlags(SecurityPermission.EXECUTION|SecurityPermission.SKIP_VERIFICATION);
assert ¬s.contains(p);
```

Fig. 3. An example demonstrating multi-dependent inspector methods and quantification over object states

Consistency of the verification logic For each inspector method, an axiom is added to the verification logic that defines the corresponding function symbol. We restrict inspector method bodies to be of the form { **return** E; }, so that we can translate an inspector method whose body is { **return** E; } into an axiom

$$\textbf{axiom}\ (\forall x_1, \ldots, x_n \bullet C\_m(x_1, \ldots, x_n) = [[E]]);$$

A potential problem with this approach is that the resulting set of axioms may be inconsistent. Notice that this is the case only if there is an inspector method call that does not terminate. To ensure that inspector methods always terminate, we restrict which method calls may appear inside inspector method bodies. Specifically, if a method call $o.m_1(\ldots)$ appears in the body of an inspector method m_2, then m_1 must be an inspector method and the the declaring class of m_2 must transitively *import* the static type of o. A type may import another type explicitly using an **import** declaration. Also, when a class C implements an interface I, I implicitly imports C (sic). For example, in Figure 3, *PermissionSet* imports *Permission* and *Permission* imports *SecurityPermission*. (We discuss the import relation in the presence of subclassing in Section 6.) It is checked at load time that the import relation is acyclic. This ensures that inspector method calls always terminate.

Quantification over object states Typically, a postcondition of a non-inspector method fully specifies the post-state of each object o that is modified by the method. It does so by specifying the value of each inspector method call on o. If an inspector method has parameters, this requires quantifying over the possible argument values. For example, in the *IntList* example (Figure 2), both the constructor and the *Add* method have an **ensures** clause that quantifies over a range of integers, to serve as the argument to the *getItem* inspector method.

If an inspector method takes an object reference as an argument, one may quantify over object references, using the following syntax:

$$\textbf{forall}\{T\ o;\ E\}$$

Variable o ranges over both unallocated and allocated objects, so that if the inspector method whose value is being defined by the quantification is called with an argument object that is allocated after the quantification is asserted, the quantification applies to that call as well. However, expression E is not allowed to access the state of o and E cannot assume the validity of o; therefore, o cannot be passed as an argument for a **state** parameter.

To support full specification of the abstract state of an object that has inspector methods that take object states as arguments, we support quantification over object states, using the following syntax:

$$\textbf{forall}\{\textbf{state}\ T\ o;\ E\}$$

Variable o again ranges over both unallocated and allocated object references, but when the state of o is inspected, it is not looked up in the heap; rather, the

state, too, is considered to be bound by the quantification, and it ranges over all possible valid object states. That is, the above quantification translates into the verification logic as follows:

$$(\forall o, o_{\text{state}} \bullet o_{\text{state}}[inv] \Rightarrow [[E]])$$

For example, referring to Figure 3,

$$\textbf{forall}\{\textbf{state } \mathit{Permission}\ q;\ \mathit{contains}(q) = \ldots\}$$

is encoded as

$$\begin{array}{l}(\forall\ q, q_{\text{state}} \bullet q_{\text{state}}[inv] \Rightarrow \\ \quad \mathit{PermissionSet\_contains}(\mathit{this}, \mathit{Heap}[\mathit{this}], q, q_{\text{state}}) = \ldots)\end{array}$$

5 Formal details

Well-formedness conditions Method preconditions and postconditions and object invariants must be *pure expressions*, and the body of an inspector method must be of the form { **return** E; } where E is a pure expression. A pure expression is a Java expression that does not contain object or array creations, simple or compound assignments, or increment or decrement operators, and that calls only inspector methods. This ensures that evaluation of pure expressions has no side-effects and that they can be translated easily into first-order logic.

Additionally, object invariants and derived invariants may depend only on the fields of **this** and on the fields of objects transitively owned by **this**. An object invariant cannot assume that **this** is valid; therefore, it cannot include inspector method calls on **this**. It can, however, include inspector method calls on objects pointed to by **rep** fields of **this**. Also, as stated before, inspector method bodies may depend only on the fields of the receiver object, objects passed as arguments for parameters marked **state**, or objects transitively owned by these objects.

Soundness of derived invariants The declaration of a derived invariant in a class C generates a proof obligation saying that it holds for all valid objects of class C. For discharging this obligation, one may assume that all derived invariants applicable to the owned objects of o hold for those objects (even if some of the owned objects are themselves of class C). To show that this proof rule is sound, we prove the following theorem:

Theorem 2 (Derived invariants). *In each execution state, for each valid object, each derived invariant applicable to it holds.*

Proof. By induction on the length of the execution. The only interesting case is when the last operation is a **pack** o; operation. The induction hypothesis allows us to apply the aforementioned proof obligation to prove that all derived invariants applicable to o hold.

Dynamically bound inspector method calls An inspector method call may be either statically or dynamically bound. For example, calls of inspector methods of final classes are statically bound, and calls of inspector methods of interfaces are dynamically bound. (In the presence of subclassing, methods may generally be called both ways. We discuss subclassing in Section 6.)

In contrast with a statically-bound inspector method, the function for a dynamically-bound inspector method is not defined once and for all by the method's declaration; rather, it is partially defined by each non-abstract inspector method that overrides it. Specifically, for each non-abstract inspector method m in class C, and each dynamically-bound inspector method m in class or interface T that it overrides, an axiom is generated that says that if the target object's type is C, both functions coincide. Formally:

$$\forall o, o_{\text{state}}, a_1, \ldots, a_n \bullet \mathsf{type}(o) = C \Rightarrow \\ T\_m(o, o_{\text{state}}, a_1, \ldots, a_n) = C\_m(o, o_{\text{state}}, a_1, \ldots, a_n)$$

6 Subclassing

In this section, we describe an extension of our approach that allows a superclass to hide its internal representation not only from client code, but from code in subclasses as well.

Our approach is based on the observation that if we want to abstract superclass state from subclasses, then objects are no longer the unit of abstraction. Indeed, we must subdivide an object along the classes that contribute state to it. We call each such subdivision an *object frame* (or *frame* for short). That is, a frame (or frame reference) is a tuple (o, C) consisting of an object reference o and the name of a class C of which o is an instance. If a class D extends a class C which extends class *Object*, then an object whose type is D consists of three frames: (o, D), (o, C), and (o, Object).

We update our heap representation accordingly. Rather than mapping object references to object states, in our new encoding the heap maps *frames* to *frame states*. The frame state for a frame (o, C) consists of the values of the fields of o declared in C.

In previous sections, we achieved abstraction of object state from client code by introducing per-object *inv* and *committed* fields, introducing **pack** and **unpack** operations on objects, introducing an ownership relation between objects, caching owned object state in owner objects, and passing object states as arguments in inspector method function applications. Completely analogously, to achieve abstraction of superclass state from subclass code, we introduce per-*frame inv* and *committed* fields, **pack** and **unpack** operations on *frames*, and an ownership relation between *frames*, and we cache owned frame state in owner frames and we pass frame states as arguments in inspector method function applications. Note that in the absence of subclassing, we again obtain the approach of the previous sections as a special case.

```
class Cell {
  int x;
  invariant 0 ≤ x;
  inspector int getX() { return x; }
  derived_invariant 0 ≤ getX();
  dynamic_invariant 0 ≤ getX();

  Cell(int x)
    requires 0 ≤ x;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  { this.x := x; pack this; }

  void setX(int x)
    requires ¬committed ∧ inv;
    requires 0 ≤ x;
    modifies this.∗;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  {
    unpack this;
    this.x := x;
    pack this;
  }
}
```

```
class MyCell extends Cell {
  invariant 1 ≤ super.getX();
  inspector int getX()
  { return super.getX() − 1; }

  MyCell(int x)
    requires 0 ≤ x;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  { super(x + 1); pack this; }

  void setX(int x)
    requires ¬committed ∧ inv;
    requires 0 ≤ x;
    modifies this.∗;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  {
    unpack this;
    super.setX(x + 1);
    pack this;
  }
}
```

Fig. 4. Example illustrating the approach for verification of programs with subclassing. An instance of type *Cell* may hold an arbitrary nonnegative integer value, which may be retrieved using inspector method *getX* and set using method *setX*. Both class *Cell* and class *MyCell* implement the *Cell* type. Class *Cell* does so by storing the value in a field x, whereas *MyCell* does so by storing the value plus one in its superclass frame. While contrived, this example shows how our approach supports the clean separation of the two aspects of subclassing: interface re-implementation and implementation inclusion. There need be no relationship between how the superclass and the subclass implement the superclass's interface; in particular, if the subclass chooses to re-use the included superclass implementation, it need not do so by direct delegation. This clean separation promotes modularity, i.e. separate development and evolution of superclass and subclass code. Note: all methods in this example are virtual and subclass methods override the corresponding superclass methods.

Frame ownership The definition of the ownership relation between frames involves a few design issues. One is: are there ownership relationships between the frames of a given object? In order to allow the subclass object invariant and subclass inspector methods to depend on superclass state, we consider a valid subclass frame to own its superclass frame. In fact, each class other than *Object* gets a special **rep** field called *super* that refers to the superclass frame. (Note that this field is exceptional in that it contains a frame reference instead of an object reference.) It follows that packing a subclass frame requires that the superclass frame is valid and uncommitted and commits it. Also, if the most derived frame of an object (i.e., the frame corresponding to the object's run-time type) is valid, it transitively owns all of the object's frames and encapsulates the entire object's state.

There is one other design decision involving the ownership relation between frames. Specifically, if a **rep** field $o.f$ declared in a class C of a valid object o points to an object p, which ownership relationship does this give rise to? In order to allow the object invariant and the inspector methods of class C to perform dynamically-bound inspector method calls on p, which access p's most derived frame, we consider frame (o, C) to own p's most derived frame. Due to the previous design decision, it follows that (o, C) transitively owns all of p's frames.

Static and dynamic binding As pointed out earlier, in general inspector method calls may be statically bound or dynamically bound. Calls of abstract methods are always dynamically bound and calls of private methods are always statically bound, but for a non-abstract inspector method of a non-final class, some calls that resolve to such a method at compile time may be statically bound and some may be dynamically bound. Since at run time, these calls may bind to different methods and yield different results, we encode them differently in the verification logic. Specifically, for calls resolved at compile time to a method m of a class or interface T, we encode statically bound ones as $T\_m(\ldots)$ and dynamically bound ones as $T\_m_{\mathrm{D}}(\ldots)$. Function symbol T_m is defined fully by the body of method m in T; function symbol T_m_{D} is defined partially by each non-abstract method that overrides it, as explained earlier.

In the encoding of a statically bound call to an inspector method m of class C on an object o, the frame state passed as the state of the receiver is always the state of frame (o, C), whereas in the encoding of a dynamically bound call, the frame state passed is always the state of the most derived frame. (Also, an argument for a parameter marked **state** in a call of a multi-dependent inspector method is always interpreted as the most derived frame.)

The object invariant and the derived invariants declared in a class C apply to frames (o, C). Since they must depend only on the state of the frame to which they apply (plus transitively owned frames), they must not include dynamically bound inspector method calls on **this**. Therefore, we always interpret inspector method calls on **this** in object invariants and derived invariants as statically bound calls.

Inspector method calls may occur in preconditions of inspector methods and in preconditions and postconditions of non-inspector methods. It is crucial for the soundness of verification that there be no confusion as to whether these calls are statically or dynamically bound. Our approach is sound regardless of which choice is made, so long as the choice is the same when verifying the caller and when verifying the callee. Note in this regard that it is fine, and often appropriate, to make different choices for statically and dynamically bound calls of the method in whose contract the inspector method call appears. This is sound so long as it is verified that the contract for dynamically bound calls is implied by the contract for statically bound calls under the assumption that the run-time type of the receiver equals its static type. The method body need then only be verified against the contract for statically bound calls.

The question remains as to how the choice of static or dynamic binding of inspector method calls in method contracts is made. Java specifies that a call in program code is dynamically bound unless the call is a **super** call or the method being called is private. However, this approach is not always appropriate for inspector method calls in method contracts. For example, consider the $getX()$ call in the contract of method *setX* of class *Cell* in Figure 4. Interpreting the call as dynamically bound would be approriate for dynamically bound calls of *setX*, but not, for example, for the **super**.*setX* call that occurs in class *MyCell*. Indeed, **super**.$setX(5)$ ensures **super**.$getX() = 5$, not $getX() = 5$.

Therefore, we interpret calls in contracts differently from calls in program code. Also, the interpretation is different depending on whether the contract is used for a statically bound call or a dynamically bound call. The rule is as follows: all calls are bound dynamically, except for **super** calls, calls of private methods, or calls on **this** in contracts for statically bound calls. Calls whose target expression is not **this** (explicitly or implicitly) or calls on **this** in contracts for dynamically bound calls are bound dynamically. For example, for the purpose of verifying the **super**.*getX* call in Figure 4, the *getX* call in the contract of *Cell.setX* is interpreted as a statically bound call. Note that this rule ensures that the contract for dynamically bound calls is equivalent to the contract for statically bound calls under the assumption that the receiver's run-time type is equal to its static type.

The import relation In the presence of subclassing, the import relation which we use to ensure termination of inspector method calls, and consistency of their axiomatization, must be refined. In particular, for each class, we introduce two nodes in the import graph, which we shall call the static node and the dynamic node. An interface only has a dynamic node. The rules are as follows:

- The dynamic node of C imports the static node of C.
- If a class C declares an **import** T; declaration, this means the static node of C imports the dynamic node of T.
- If a class D extends a class C, this means the static node of D imports the static node of C, and the dynamic node of C imports the static node of D.
- If a class C implements an interface I, this means the dynamic node of I imports the static node of C.

- If a class C declares an inspector method whose body includes a dynamically bound inspector method call resolved at compile time to a method of a type T, then the static node of C must transitively import the dynamic node of T.
- If a class C declares an inspector method whose body includes a statically bound inspector method call declared by a class D, then the static node of C must import the static node of D.

This ensures that there can be no inspector method call cycles within an inheritance hierarchy.

Dynamic invariants As stated above, a class C may declare a derived invariant, using the **derived_invariant** keyword. This invariant applies to frames (o, C) only, and inspector method calls in derived invariants are always interpreted as statically bound calls. This means that derived invariants declared by a class C are useful for a subclass of C when performing **super** calls, or if C is a **final** class, but not for client code that accesses an instance of C through dynamically bound calls.

To convey properties of and relationships between dynamically bound inspector methods, a class or interface may declare one or more *dynamic invariants*, using the **dynamic_invariant** keyword. A dynamic invariant declared by a type T is enforced against all non-abstract classes that implement or extend T. It follows that a dynamic invariant declared by a type T holds for all instances of T.

Note: dynamic invariants do not subsume derived invariants; for example, an abstract class that implements some of its inspector methods may declare a derived invariant to specify a relationship between the inspector methods it implements.

Method inheritance As in the Boogie methodology [1], we do not allow method inheritance as such. That is, each class must override all visible methods of its superclass. This rule is crucial for the soundness of our approach since our approach is based on the assumption that if a dynamically bound call binds to a method m declared by a class C, then the static type C of m's receiver is equal to its run-time type. This is true only if m was not inherited from C by the receiver's run-time type.

However, to reduce programming overhead, we follow the Boogie methodology in generating a default override for each method that is not overridden explicitly. The body of the default override for a method m is of the form

$$\{\ \textbf{unpack this};\ \textbf{super}.m();\ \textbf{pack this};\ \}$$

(or similar if m has parameters or a return value). Note that default overrides are subject to verification just like explicitly declared methods. If a default override fails to verify, an explicit override must be provided.

7 Related Work

The Boogie methodology Our approach is based for object invariants, ownership, and method framing on the Boogie methodology [1]. In order to add support for inspector methods, we applied the following modifications:

- In the Boogie methodology, the heap is encoded in the verification logic as a function that maps an object reference and field name combination to a field value. We encode the heap as a function from frame references to frame states. (Note: the notion of object frames was introduced in [1], under the name *class frames*, but in [1] it was not used as the basis for the encoding of the heap.)
- We extend the semantics of the **pack** command to store a copy of each owned frame's state in a special field of the owner frame. This allows inspector method calls to be encoded as function applications that take object frame states.
- In the Boogie methodology, method frame conditions are expressed as equalities between field values. We express method frame conditions as equalities between object frame states. Together with the previous modification, this enables a theorem prover to carry information regarding inspector method call return values across method calls.
- Instead of introducing a single *committed* bit and a single *inv* field per object, and storing in the *inv* field the name of the most derived type whose object frame is valid, we introduce separate *inv* and *committed* bits in each object frame.

Method calls in specifications Darvas and Müller [3] identify and propose solutions for problems that arise when method calls are used in specifications. Specifically, the authors show how to deal with abrupt termination, object creation, and inconsistent axiomatization due to unsatisfiable postconditions. Methods called in specifications must be pure, which means they do not modify existing objects. Inspector methods are like pure methods, with the additional constraint that they depend only on the state of the receiver object and objects passed as arguments for parameters marked **state** (and their transitively owned objects). A minor difference is that we do not allow inspector methods to declare a postcondition and that we derive the axiom that defines the corresponding function from the inspector method's body rather than its postcondition. We avoid the abrupt termination issue by verifying that the inspector method body does not throw any exceptions.

Darvas and Müller's solution to the object creation issue is to pass the heap as an argument to the function and have the function return a new heap together with its result value. We did not adopt this solution because it is incompatible with our approach to framing; specifically, our approach requires that only the state of the dependee objects is passed to the inspector method's function, rather than the entire heap. Therefore, we disallow object creation in inspector methods.

We avoid the problem of inconsistent axiomatization by imposing a partial order on classes and by allowing nested inspector method calls to proceed along this partial order only. This ensures that inspector method calls always terminate and have a return value under Java semantics, and the return values thus obtained always satisfy the system of equations that defines the inspector method functions of a given program.

State abstraction in ownership systems Müller [7] combines a notion of *model fields* with an ownership type system called *Universes*. Model fields are comparable with parameterless inspector methods. The Universes ownership system is less flexible than that of the Boogie methodology; for example, it does not allow ownership transfer. On the other hand, Müller allows model fields of an object o to depend on fields of *peer objects* of o, i.e. objects that have the same owner as o, provided that the model field definitions are visible to the field declarations, i.e. both are in the same module. We intend to add support for visibility-based inspector methods to our approach as future work.

Leino and Müller [6] achieve state abstraction by combining the Boogie ownership system with another notion of model fields, similar to but distinct from that of Müller [7]. Model fields in [6] are encoded in the verification logic as if they are stored in the heap along with concrete fields. Each model field declaration specifies a model field constraint that serves as an abstraction relation. The constraint for a model field $o.m$ needs to hold only when o is valid. As part of executing a **pack** statement on an object o, the constraints of the model fields of o are checked and, if they do not hold, a new value that satisfies the constraint is assigned to the model field. If no such value exists, the pack statement is considered invalid.

A constraint may underspecify a model field, and subclasses may strengthen an inherited model field's constraint. As a result, packing an object for a subclass may assign a new value to a model field declared in a direct or indirect superclass. An underspecified model field is similar to an abstract inspector method with a dynamic invariant, and a strengthening of an inherited model field's constraint by a subclass is similar to an inspector method that overrides an abstract inspector method. However, "overriding" a fully specified model field with another differently fully specified model field, similar to overriding a non-abstract inspector method, is not supported in [6]. This means that a subclass is forced to adopt fully specified public superclass model fields as part of its own abstract state, without the ability to provide a different abstraction function. Alternatively, if a class leaves a model field underspecified and does not tie it to its own concrete state, so that subclasses have maximum freedom in providing an abstraction function, then the class cannot use the model field to abstractly expose its own state. Therefore, it also is not able to fully implement methods specified using the model field. As a result, in practice, classes with underspecified model fields are effectively abstract. In other words, [6] does not fully support specification of classes that may be used both for direct instantiation and as superclasses, while leaving subclasses free to provide their own abstraction functions for superclass model fields. (For example, see class *Cell* in Figure 4.)

To support the kind of specifications enabled in our approach by parameterized inspector methods, such as the *getItem* method in the *IntList* example of Figure 2, [6] would have to use model fields containing special-purpose immutable objects such as immutable list objects (known as *model types* in JML [5]).

Ownership-free approaches Kassios [4] also uses abstraction functions, but instead of an ownership system he proposes *dynamic frames* to abstractly specify an abstraction function's dependencies and a mutator method's effects. Dynamic frames are themselves abstraction functions that return sets of locations. A module specification may specify a frame (i.e., an upper bound on the set of locations that an abstraction function depends on) for each abstraction function separately.

The dynamic frames approach subsumes our approach on an abstract level. However, it is formulated in the context of an idealized logical framework; for example, it does not show how to apply the approach to Java-like inheritance. Also, the proposed approach has not been applied in the context of an automatic program verifier.

Parkinson and Bierman [9] and Parkinson [8] extend separation logic with *abstract predicates* and apply it to Java to achieve state abstraction for Java programs. Abstract predicates are similar to inspector methods that return a boolean value. However, as in the case of the aforementioned dynamic frames approach, Parkinson and Bierman do not restrict the set of locations that an abstract predicate may depend on. Rather, it is up to client code to track the separation between abstract predicates. Parkinson and Bierman solve the problem of well-definedness of abstract predicates by allowing abstract predicates to appear inside other abstract predicates only in positive positions, and by taking the least fixpoint of a set of abstract predicate definitions as their meaning. Subclassing is addressed by introducing *abstract predicate families*; that is, an abstract predicate name may be subscripted by a class name, and a separate definition may be given for each subscript. For example, the abstract predicate saying that a *Cell* instance o holds the value v could be written $cell_{\mathsf{type}(o)}(o, v)$.

The use of abstract predicates in method contracts typically requires the use of universally quantified logical variables whose scope extends across both the pre- and post-state. For example, the contract for a method that that increments the value of a cell would say that for each value v, if $cell(o, v)$ holds in the pre-state, then $cell(o, v + 1)$ holds in the post-state. This could be a disadvantage compared to model fields or inspector methods, in particular for run-time checking.

Redundant invariants Our derived invariants and dynamic invariants are similar to JML's [5] redundant invariants, in that an object's redundant invariants must be implied by its invariants. However, the interaction between JML's different kinds of invariants and JML model fields is not clear. Specifically, on which invariants is a JML model field's *represents clause*, which defines its abstraction relation, allowed to depend to prove absence of evaluation errors such as null

dereferences or divisions by zero? And which invariants are allowed to mention model fields of **this**?

8 Future Work

We are currently working on a rigorous and comprehensive formalization and soundness proof of the approach.

We are also investigating relaxations of the requirement that nested inspector method calls follow a partial order. One relaxed rule would be that for each nested call, the callee must depend on fewer objects (or object frames) than the caller. Or alternatively, that the size of the argument list, defined as the total number of object states, including duplicates and internal owned object states, must decrease.

Another area of extension is to allow inspector methods to depend on non-owned *peer* objects of classes declared in the same module.

9 Conclusion

We proposed an approach to the verification of object-oriented programs that use inspector methods for state abstraction in specifications.

We solve the problem of encoding in the verification logic whether a given method call or field assignment affects a given inspector method call's return value, by

- modeling the heap as a function that maps object references to object states,
- logically (but not physically) storing a copy of the state of owned objects in special fields of the owner object, and
- encoding inspector method calls as function applications whose arguments include the object states on which the inspector method depends.

We support multi-dependent inspector methods, i.e. inspector methods that depend on the state of objects passed as arguments. To enable the specification of the return values of all possible calls of a multi-dependent inspector method, we support quantification over object states.

Our approach to subclassing is to separate its two aspects: superclass interface re-implementation and superclass implementation inclusion. Subclasses may override superclass inspector methods; our approach preserves soundness by binding inspector method calls statically or dynamically as appropriate. A distinction is made between derived invariants, which apply only to the declaring class, and dynamic invariants, which apply to subclasses as well.

We implemented the core of our approach in a custom build of the Spec# program verifier [2]. This will allow us to gain experience and validate the approach on larger programs.

We are currently working on a rigorous and comprehensive formalization and soundness proof of the approach.

Acknowledgements

The authors thank Rustan Leino, Peter Müller, and Jan Smans for helpful comments and discussions. Bart Jacobs is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).

References

1. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004. Special issue: ECOOP 2003 workshop on FTfJP.
2. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.
3. Ádám Darvas and Peter Müller. Reasoning about method calls in JML specifications. In Francesco Logozzo, editor, *Proceedings of the Seventh Workshop on Formal Techniques for Java-like Programs (FTfJP 2005)*, 2005.
4. Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. Technical Report 528, Dept. of Computer Science, University of Toronto, July 2005.
5. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Department of Computer Science, Iowa State University, July 2005.
6. K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *Proc. ESOP*, 2006. To appear.
7. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
8. Matthew J. Parkinson. *Local reasoning for Java*. PhD thesis, Computer Laboratory, Cambridge University, 2005.
9. Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proc. POPL*, 2005.