

# Verifying Programs Using Inspector Methods for State Abstraction

*Bart Jacobs*      *Frank Piessens*

*Report CW432, December 2005*



Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Verifying Programs Using Inspector Methods for State Abstraction

*Bart Jacobs*      *Frank Piessens*

*Report CW432, December 2005*

Department of Computer Science, K.U.Leuven

## **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.

In this paper, we propose an approach to the verification of programs that use inspector methods in method contracts and object invariants. Performing state abstraction in a programming language that allows aliasing through object references poses a framing problem. Our solution to this framing problem is a formulation of the Boogie methodology in terms of *read bags and write sets* which, combined with the Boogie methodology's ownership system, abstractly capture a method's effects. We show how programs that are specified using inspector methods can be translated into the input language of the Boogie program verifier.

# Verifying Programs Using Inspector Methods for State Abstraction

Bart Jacobs and Frank Piessens

{bartj,frank}@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.

In this paper, we propose an approach to the verification of programs that use inspector methods in method contracts and object invariants. Performing state abstraction in a programming language that allows aliasing through object references poses a framing problem. Our solution to this framing problem is a formulation of the Boogie methodology in terms of *read bags and write sets* which, combined with the Boogie methodology's ownership system, abstractly capture a method's effects. We show how programs that are specified using inspector methods can be translated into the input language of the Boogie 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' 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.

```
class Cell {
  int x;
  inspector int getX()           Cell c1 := new Cell(0);
  { return x; }                 int y := c1.getX();
  Cell(int value)               assert y = 0;
  ensures getX() = value;      Cell c2 := new Cell(5);
  { x := value; }              c1.setX(10);
  void setX(int value)         assert c1.getX() = 10;
  ensures getX() = value;     assert c2.getX() = 5;
  { x := value; }
}
```

Figure 1: A class specified using an inspector method, and a client program

In this paper, we concern ourselves with how to prove the correctness of programs such as the one in Figure 1. Specifically, we wish to translate object-oriented programs with inspector methods into BoogiePL [6, 2], an abstract procedural programming language that comes with an automatic program verifier. Figures 2 and 3 show naive translations of class *Cell* and the client program from Figure 1, respectively, into BoogiePL. The program in Figure 3 does not verify because of a lack of framing.

We introduce our approach to verification of object-oriented programs with inspector methods gradually. In Section 2, we start off with Java minus exceptions, concurrency, and inheritance, plus *requires* and *ensures* clauses and inspector methods, and we describe a naive translation to BoogiePL for this simple language. In Section 3, we propose an approach to framing of methods. In Section 4, we add support for object invariants and ownership. In Section 5, we add support for inheritance. We end with a discussion, related work, future work, and a conclusion. Our approach is based on the Boogie methodology [1].

## 2 Framework

### 2.1 Source Language

We start off with a simple subset of Java, extended with simple specification constructs; in subsequent sections, we add specification and implementation features. The initial language is Java minus exceptions, concurrency, and inheritance, extended with inspector methods and simple method contracts. We also add non-null types [7] to avoid having to deal with null references pervasively.

**Inspector methods.** A method may be marked using a new keyword **inspector**. Methods so marked are *inspector methods*. An inspector method may not be static. The body of an inspector method must be of the form `{ return E; }`, where *E* is a *confined expression*, i.e. a literal, a parameter, **this**, a field read **this.f**, or a Java operator applied to confined expressions.

**Method contracts.** A method may declare a number of **requires** clauses and **ensures** clauses. The **requires** or **ensures** keyword must be followed by a Java expression that satisfies the following restrictions: it must not include object creation, array creation, assignment, increment, or decrement expressions, and it must not include calls of any methods other than inspector methods. Inspector methods may declare only **requires** clauses; they must not declare **ensures** clauses.

**Non-null types.** In order to reduce the overhead caused by the possibility of null references, we interpret reference types as non-null types by default. That is, given a class name *C*, the type *C* does not include null. Java-style nullable types are available through the *C# 2.0* syntax *C?*.

As is well-known [7], non-null fields pose special difficulties. As in *Spec#*, we require a constructor to initialize all non-null fields before it returns or leaks **this** by passing it as the receiver or as an argument to a method or constructor call, by writing it to a field, or (in the presence of inheritance) by calling the base class constructor.

### 2.2 BoogiePL

Instead of directly generating verification conditions from source programs, we follow the *Spec#* approach [2] of first translating the source program into an

```

type field(_) < name;
type s = ( $\forall T \bullet [field(T)]T$ );
type ref;
type heap = ref  $\leftrightarrow$  s;
distinct const Cell_x: field(int);
function Cell_getX(s): int;
axiom ( $\forall H$ : heap, o: ref  $\bullet$  Cell_getX( $H[o]$ ) =  $H[o, Cell\_x]$ );
var H: heap;
procedure Cell_getX(this: dom H) returns (result: int)
{
  entry :
    result :=  $H[this, Cell\_x]$ ;
    return;
}
procedure Cell_Cell_ens(oldH: heap, this: ref(oldH), value: int) returns (truth: bool)
requires dom oldH  $\subseteq$  dom H;
{
  entry :
    truth :=  $Cell\_getX(H[this]) = value$ ;
    return;
}
procedure Cell_Cell(this: dom H, value: int) modifies H
ensures  $Cell\_getX(H[this]) = value$ ;
{
  entry :
     $H[this, Cell\_x] := 0$ ;
     $H[this, Cell\_x] := value$ ;
    return;
}
procedure Cell_setX_ens(oldH: heap, this: ref(oldH), value: int) returns (truth: bool)
requires dom oldH  $\subseteq$  dom H;
{
  entry :
    truth :=  $Cell\_getX(H[this]) = value$ ;
    return;
}
procedure Cell_setX(this: dom H, value: int) modifies H
ensures  $Cell\_getX(H[this]) = value$ ;
{
  entry :
     $H[this, Cell\_x] := value$ ;
    return;
}

```

Figure 2: BoogiePL program for class *Cell* of Figure 1 (in a system without object invariants or ownership)

```

type field( $\_$ ) < name;
type s = ( $\forall T \bullet [field(T)]T$ );
type ref;
type heap = ref  $\leftrightarrow$  s;
function Cell_getX(s): int;
var H: heap;
procedure Cell_Cell(this:  $\text{dom } H$ , value: int) modifies H
  ensures Cell_getX(H[this]) = value;
procedure Cell_setX(this:  $\text{dom } H$ , value: int) modifies H
  ensures Cell_getX(H[this]) = value;
procedure Main() modifies H
{
  var c1: ref, y: int, c2: ref;
  entry :
    alloc c1 in H;
    call Cell_Cell(c1, 0);
    y := Cell_getX(H[c1]);
    assert y = 0;
    alloc c2 in H;
    call Cell_Cell(c2, 5);
    call Cell_setX(c1, 10);
    assert Cell_getX(H[c1]) = 10;
    assert Cell_getX(H[c2]) = 5;
    return;
}

```

Figure 3: BoogiePL program for the client program of Figure 1 (in a system without object invariants or ownership)

abstract procedural programming language called BoogiePL. Verification conditions are then generated from the latter program.

BoogiePL has declarative theory definition constructs, including constant, function symbol, and axiom declarations, and imperative programming constructs, including global variable and procedure declarations. Procedure bodies consist of a number of local variable declarations and a number of blocks. Each block has a unique label. The block further consists of a sequence of commands, terminated by a special transfer command. A transfer command is either a return command, or a non-deterministic goto to one or more blocks. Commands include assignments, assertions, assumptions, and procedure calls.

We refer the reader to [6] for a complete reference on BoogiePL.

In this paper, we introduce a few extensions to BoogiePL in order to make BoogiePL programs more readable.

**Arrays** We have single-dimensional arrays. Arrays can have any type as their index type and any type, including an array type, as their element type. We perform automatic currying of array element reads and writes, which yields the effect of multidimensional arrays.

**Fields** To avoid excessive casting, we allow parameterized types. This can be used to parameterize the type of field names by the type of their values.

Note that since verification condition generation erases all types, we still need to be able to compare field names with distinct element types; therefore, we declare the type of field names to be a subtype of the type of names.

**Allocation** BoogiePL has global variables and local variables, but it has no built-in notion of a heap or dynamic allocation. One can model a heap as a global variable containing an array that maps a reference to a tuple containing an *alloc* bit and an object state. Allocation of a new object means choosing an arbitrary reference that maps to a tuple whose *alloc* bit is false, and then setting the *alloc* bit to true.

In order to be able to prove that a newly allocated object is distinct from existing objects, one needs the following:

- Procedure preconditions that say that arguments are allocated
- Axioms that say that field values of allocated objects are allocated
- Procedure postconditions that say that all objects that were allocated in the pre-state are still allocated in the post-state
- Procedure postconditions that say that return values are allocated

To minimize the overhead imposed by these requirements on the BoogiePL examples in this paper, we extend BoogiePL with support for a certain kind of *partial arrays* that have the notion of a *domain*. We assume these to be *monotonic*, in the sense that the domain only ever grows. That is, each command that assigns a new value to a variable whose type is a partial array type, is augmented with an assumption that says that the domain of the new value includes the domain of the old value. A partial array mapping a type  $\rho$  to a type  $\sigma$  is denoted as  $\rho \hookrightarrow \sigma$ . We allow  $\text{dom } H$ , where  $H$  is a variable of partial array type that is in scope, to be used as a type. These types are translated into assumptions as part of verification condition generation.

Note, however, that it is not an error to index a partial array outside its domain.

To allocate a new element from the domain, we introduce the new command **alloc  $x$  in  $H$** .

Note that the *alloc* bit in a partial array is implicit; it cannot be mentioned explicitly. The partial array maps indices directly to the declared element type.

**The Null Reference** The type *ref* does not include *null*. There is a distinct type *refnonnull* that is a supertype of *ref*. *null* is declared to be of type *refnonnull*.

## 2.3 Verifying the Validity of Method Contracts

The expressions in **requires** and **ensures** clauses are interpreted strictly according to Java semantics. If, according to Java semantics, evaluation of an expression causes an exception to be thrown, the method contract is considered to be invalid. That is, it is the responsibility of the author of the method contract to ensure its evaluation never throws an exception, in any pre-state or post-state.

To verify the validity of a method contract, for each precondition and for each postcondition, we generate a procedure in the BoogiePL program that evaluates the expression. We will refer to procedures generated to verify the validity of specifications as *specification validity procedures*, to distinguish them from *code validity procedures* generated from method bodies. The translation process is the same as for an expression in program code: **assert** commands are inserted to assert validity constraints such as array index constraints and null pointer constraints.

Unique for the translation of postconditions is that they are two-state predicates. Specifically, they are evaluated in the post-state but they may include **old** expressions that refer to the pre-state. Whereas the post-state is passed to the procedure as the value of the global variable *H*, the pre-state is passed as an explicit parameter *oldH*.

The procedure for the precondition does not have a precondition itself. The procedure for the postcondition receives the method's precondition as its precondition. The latter is translated as a predicate on *oldH*.

## 2.4 Translation of Inspector Methods

There are two aspects to the translation of inspector methods: translation of the methods themselves, and translation of calls of inspector methods.

As is the case for other methods, for each inspector method a BoogiePL procedure is generated. Contrary to procedures for non-inspector methods, this procedure is never called from other procedures; it serves only to verify the validity of the inspector method's body. For example, a potential division by zero in the inspector method's body would be caught through the verification of this procedure.

Contrary to other methods, inspector methods generate a BoogiePL function declaration in addition to a procedure. The function is used to translate calls of the method that occur in program code or in method contracts. The function takes an object state. The function declaration is accompanied by an axiom that defines the function's value for a given object state. The right-hand side of the definition is a translation of the inspector method's body.

# 3 Framing

**The Framing Problem** The program in Figure 1 is valid. However, its translation into BoogiePL, as given in Figures 2 and 3, does not verify. Specifically, the final **assert** command in procedure *Main* fails. The failure is caused by the final call of procedure *Cell\_setX*. Whereas the postcondition of the preceding call (of *Cell\_Cell*) establishes  $Cell\_getX(H[c2]) = 5$  for its post-state and the

pre-state of the call of *Cell\_setX*, this latter call does not allow that information to carry over to its post-state. All *Cell\_setX*'s contract says is that *H* is modified to some arbitrary new value that satisfies  $Cell\_getX(H[c1]) = 10$ . The contract of *Cell\_setX* specifies the new state of its receiver object, but it fails to specify that the state of all other objects remains unchanged. This is known as a *frame condition*.

**The Write Set** To solve the framing problem, we extend the specification language so that methods may specify a frame for their effects. We also introduce new validity constraints to enforce this frame.

Just for specification and enforcement of frames, we introduce a new piece of global state, in the form of a global variable *W*, called the *write set*. The write set is the set of objects that the program may modify at any one point in time. Specifically, an assignment to a field *o.f* is subject to the validity constraint that  $o \in W$ . Creating an object automatically adds it to the write set.

**Method Contracts** We extend the syntax of **requires** and **ensures** clauses in the source language so that they may state conditions on the write set. Specifically, we add the new keyword **writable** and the new expression form **writable**(*E*), called a *writability condition*, where *E* is an expression that denotes an object. **writable**(*o*) means that *o* is in the write set. The new expression form may be used only in contracts; there is no way to inspect the write set in program code. It follows that the behavior of a valid method does not depend on the write set.

Some ways of using **writable**(*o*) in method contracts are not meaningful. For example, writing **writable**(*o*)  $\vee$  **writable**(*p*) in a precondition is not useful since the method will be able to modify neither *o* nor *p*. Another example is **writable**(*o*)  $\Rightarrow x = 5$  since the method will not be able to establish the antecedent. Therefore, we require that the new expression form appear only in *definite positions*. Consider an expression *C* with a hole [] in it. We define *definite C* as below, where *E* and *E'* are arbitrary expressions (which may contain writability conditions). Note that in this text we use  $\wedge$  as a nicer symbol for Java's shortcut conjunction **&&**, and we use  $\vee$  as a nicer symbol for Java's shortcut disjunction **||**.

$$\begin{aligned} & \text{definite } [] \\ \text{definite } C & \Rightarrow \text{definite } (C \wedge E) \\ \text{definite } C & \Rightarrow \text{definite } (E \wedge C) \\ \text{definite } C & \Rightarrow \text{definite } (E \Rightarrow C) \\ \\ \text{definite } C & \Rightarrow \text{definite } (E \vee C) \\ \text{definite } C & \Rightarrow \text{definite } (E ? C : E') \\ \text{definite } C & \Rightarrow \text{definite } (E ? E' : C) \end{aligned}$$

The shortcut behavior of  $\vee$  is the reason why we allow  $E \vee C$  but not  $C \vee E$ .

A precondition or postcondition *P* containing writability conditions is valid if for each *C* and *E* such that  $P = C[\text{writable}(E)]$ , it holds that *definite C* (where  $C[X]$  means *C* with *X* substituted for the hole).

The value of an expression that includes writability conditions is uniquely determined by the values of the variables (e.g. parameters or local variables, as applicable) used in the expression, the values of fields read in the expression (i.e. the heap), and the current write set. If writability conditions appear only in definite positions, we have the property that given a fixed variable valuation and heap, if a predicate (i.e. boolean expression) *P* evaluates to *true* under a given write set, it also evaluates to *true* under each greater write set. In fact,

the definite positions constraint ensures that if under a given variable valuation and heap, the expression is satisfiable at all, then there is a minimal write set so that it evaluates to true. We call this the predicate's *required write set*  $W_P$  for the given argument list and heap.

Moreover, we can easily derive an expression for  $W_P$  from that of  $P$  syntactically. Specifically, let  $o$  be a fresh variable. For a given  $P$ , we define  $P_W[o]$  as follows:

$$P_W[o] = \neg(P[(o \neq E)/(\mathbf{writable}(E)]_{\forall E})$$

We then have the following:

$$o \in W_P \Leftrightarrow P_W[o]$$

That is, if we replace each writability condition  $\mathbf{writable}(E)$  in  $P$  by  $o \neq E$ , obtaining  $P'$ , then  $\neg P' = (o \in W_P)$ .

*Proof.*  $P$  is equivalent to an expression of the following form:

$$(P_1 \Rightarrow \mathbf{writable}(E_1)) \wedge \dots \wedge (P_n \Rightarrow \mathbf{writable}(E_n)) \wedge P_0$$

where the  $P_i$  and  $E_i$  contain no writability conditions. When instantiated with a specific heap and argument list, we obtain either *false* or the following:

$$\mathbf{writable}(o_1) \wedge \dots \wedge \mathbf{writable}(o_k)$$

. If we do the same with  $P_W[o]$ , we get

$$\neg(o \neq o_1 \wedge \dots \wedge o \neq o_k)$$

□

**Solving the Framing Problem** We are now ready to solve the framing problem. Our solution exploits a property of the semantics of BoogiePL: while at any point in time, the write set of a program in our source language contains all objects ever created by the program, this is a property that can be derived only by looking at whole programs; BoogiePL semantics, however, prescribes *modular verification*. BoogiePL always assumes that a program presented to it is a *partial* program, and it verifies the validity of the program in all possible contexts.

It follows that the only information a method has about the write set in its pre-state, is what is required by its precondition. At any time, a method may modify only those objects which *it knows* are in the write set. We call the set of those objects *the method's write set*. Initially, a method's write set is equal to its precondition's required write set  $W_P$ , which we also call *the method's required write set*. A method's write set may evolve during the course of its execution as the result of object creations and method calls. When a method creates an object, the object is added to the method's write set. When a method calls another method, the call is only valid if the callee's required write set is included in the caller's write set. The call affects the caller's write set as follows: first, the callee's required write set is subtracted from the caller's write set. When the callee returns, the callee's *ensured write set* (the write set  $W_Q$  required by the callee's postcondition  $Q$ ) is added to the caller's write set. When a method returns, its write set must include the write set required by its postcondition, i.e. the method's ensured write set.

One property of a method's write set is that it always contains only allocated objects, since the precondition can only mention allocated objects. Another property is that if in the pre-state of a call, an allocated object is not in the callee's required write set, then the call does not modify the object. Indeed,

```

class Cell {
  int x;
  inspector int getX()
  { return x; }
  Cell(int value)
    ensures writable(this) ∧ getX() = value;
  { x := value; }
  void setX(int value)
    requires writable(this);
    ensures writable(this) ∧ 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;

```

Figure 4: A class specified using an inspector method, and a client program

the object will never be in the write set of the callee or of any of its transitive callees. Proof: by induction on the length of a method execution.

The latter property effectively frames the effects of a call. We may assume it at any call site. In our translation to BoogiePL, we add this property as a *free ensures clause* to each method’s procedure. (A free ensures clause is an ensures clause that is assumed without proof.) We call this ensures clause the method’s *frame condition*. Specifically, the frame condition for a method with precondition  $P$  is as follows:

$$\text{free ensures } (\forall o \in \text{old}(\text{dom } H) \bullet \neg \text{old}(P_W[o]) \Rightarrow H[o] = \text{old}(H[o]));$$

Additionally, since we introduced a new global variable  $W$ , we need to frame the method’s effect on this variable as well. One can prove by induction on the length of a method execution that an object that is writable in the pre-state and that is not in the required write set, remains writable:

$$\text{free ensures } (\forall o \in \text{old}(\text{dom } H) \bullet \text{old}(o \in W \wedge \neg P_W[o]) \Rightarrow o \in W);$$

Figure 4 shows a version of the program of Figure 1 whose method contracts include writability conditions. The translation is shown in Figures 5 and 6. Thanks to the frame conditions, the program now verifies.

## 4 Object Invariants and Ownership

In this section, we extend the specification formalism and the verification approach with support for object invariants and ownership, essentially as in [1], as well as inspector methods with parameters and preconditions. The example in Figure 7 motivates and illustrates the approach.

An object of class *IntList* in Figure 7 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.

### 4.1 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*. It would be unfortunate to require the method’s caller to guarantee this; this

```

type field⟨_⟩ < name;
type s = (∀T • [field⟨T⟩]T);
type ref;
type heap = ref ↦ s;
distinct const Cell_x: field⟨int⟩;
function Cell_getX(s): int;
axiom (∀H: heap, o: ref • Cell_getX(H[o]) = H[o, Cell_x]);
var H: heap;
var W: [ref]bool;
procedure Cell_getX(this: dom H) returns (result: int)
{
  result := H[this, Cell_x];
  return;
}
procedure Cell_Cell_ens(oldH: heap, this: ref(oldH), value: int) returns (truth: bool)
requires dom oldH ⊆ dom H;
{
  truth := this ∈ W ∧ Cell_getX(H[this]) = value;
  return;
}
procedure Cell_Cell(this: dom H, value: int) modifies H, W
requires this ∈ W;
ensures this ∈ W ∧ Cell_getX(H[this]) = value;
free ensures (∀o ∈ old(dom H) • o ≠ this ⇒ H[o] = old(H[o]));
free ensures (∀o ∈ old(dom H) • old(o ∈ W ∧ o ≠ this) ⇒ o ∈ W);
{
  H[this, Cell_x] := 0;
  assert this ∈ W;
  H[this, Cell_x] := value;
  return;
}
procedure Cell_setX_req(this: dom H, value: int) returns (truth: bool)
{
  truth := this ∈ W;
  return;
}
procedure Cell_setX_ens(oldH: heap, this: ref(oldH), value: int) returns (truth: bool)
requires dom oldH ⊆ dom H;
{
  truth := this ∈ W ∧ Cell_getX(H[this]) = value;
  return;
}
procedure Cell_setX(this: dom H, value: int) modifies H, W
requires this ∈ W;
ensures this ∈ W ∧ Cell_getX(H[this]) = value;
free ensures (∀o ∈ old(dom H) • o ≠ this ⇒ H[o] = old(H[o]));
free ensures (∀o ∈ old(dom H) • old(o ∈ W ∧ o ≠ this) ⇒ o ∈ W);
{
  assert this ∈ W;
  H[this, Cell_x] := value;
  return;
}

```

Figure 5: BoogiePL program for class *Cell* of Figure 4 (in a system without object invariants or ownership)

```

type field⟨_⟩ < name;
type s = (∀T • [field⟨T⟩]T);
type ref;
type heap = ref ↦ s;
function Cell_getX(s): int;
var H: heap;
procedure Cell_Cell(this: dom H, value: int) modifies H, W
  requires this ∈ W;
  ensures this ∈ W ∧ Cell_getX(H[this]) = value;
  free ensures (∀o ∈ old(dom H) • o ≠ this ⇒ H[o] = old(H[o]));
procedure Cell_setX(this: dom H, value: int) modifies H, W
  requires this ∈ W;
  ensures this ∈ W ∧ Cell_getX(H[this]) = value;
  free ensures (∀o ∈ old(dom H) • o ≠ this ⇒ H[o] = old(H[o]));
procedure Main() modifies H, W
{
  var c1: ref, y: int, c2: ref;
  alloc c1 in H;
  W[c1] := true;
  call Cell_Cell(c1, 0);
  y := Cell_getX(H[c1]);
  assert y = 0;
  alloc c2 in H;
  W[c2] := true;
  call Cell_Cell(c2, 5);
  call Cell_setX(c1, 10);
  assert Cell_getX(H[c1]) = 10;
  assert Cell_getX(H[c2]) = 5;
  return;
}

```

Figure 6: BoogiePL program for the client program of Figure 4 (in a system without object invariants or ownership)

would cause the caller to depend on the internals of class *IntList*. To solve this problem, we provide a mechanism called *object invariants* that allows a developer to expose conditions on internal state to clients in an *abstracted* form. Specifically, we allow a developer to declare an object invariant using the new **invariant** keyword, and in each object, we introduce a special boolean field *inv* and we restrict 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.

In the absence of ownership, the object invariant may be any *confined expression*, as defined in Section 2.1.

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*;. In the absence of ownership, these statements behave as follows:

<b>pack</b> <i>o</i> ; $\equiv$	<b>unpack</b> <i>o</i> ; $\equiv$
<b>assert</b> <b>writable</b> ( <i>o</i> );	<b>assert</b> <b>writable</b> ( <i>o</i> );
<b>assert</b> $\neg o.inv$ ;	<b>assert</b> <i>o.inv</i> ;
<b>assert</b> <i>Inv</i> ( <i>o</i> );	<i>o.inv</i> := <i>false</i> ;
<i>o.inv</i> := <i>true</i> ;	

That is, the **pack** *o*; operation checks that *o* is writable and mutable, and that *o*’s object invariant holds. It then marks the object as valid. The **unpack** *o*; operation checks that *o* is writable and 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 method bodies are verified under the assumption that **this** is valid. An inspector method call generates a proof obligation that the target object is valid.

In addition to an object invariant, a class may declare a *derived invariant*, using one or more **derived invariant** declarations. These are similar to JML’s [9] redundant invariants. A class that declares a derived invariant 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**.

## 4.2 Ownership

Does the client program provided in Figure 7 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 2.4, the BoogiePL function generated for *getItem* takes only the state of **this**, not the state of the *elems* array, as a parameter.

Indeed, *getItem*'s return expression is not a confined expression as defined in Section 2.1. In order to allow inspector methods like *getItem*, we introduce an *ownership system*, where an object can *own* other objects. A property enforced by the ownership system is that modifying an object requires modifying its transitive owner objects. This makes it possible for the BoogiePL function for an inspector method to continue to take just the state of the receiver object as its argument, while depending on the state of objects transitively owned by the receiver object as well.

In the absence of ownership, the example would be broken in a second way as well. Specifically, the *Add* method requires only the receiver to be writable. However, it performs an assignment to the *elems* array. It cannot require the *elems* array to be writable in its precondition, since this would expose clients to the internal structure of the class. The ownership system solves this problem as well, as follows. Unpacking an object causes that object's owned objects to become writable. Packing an object causes its *rep* objects to become no longer writable. Therefore, in order to modify an owned object, its transitive owners must first be unpacked.

This allows us to extend the definition of confined expressions as follows. A confined expression is a literal, **this**, a parameter, a field read **this.f**, an inspector method call **this.m**( $E_1, \dots, E_n$ ), an owned object field read **this.g.f**, an owned object inspector method call **this.g.m**( $E_1, \dots, E_n$ ), or a Java operator applied to confined expressions. In the above, *g* must be a **rep** field and  $E_1, \dots, E_n$  must be confined expressions. Note that array length expressions and array element expressions are treated like "inspector method calls" on the array.

Object invariants, inspector method bodies, and inspector method preconditions must be confined expressions. This ensures that their value depends only on the state of objects reflexively-transitively owned by **this**.

Object invariants and inspector method bodies may not contain inspector method calls on **this**. This ensures that inspector method calls always terminate. It also ensures that inspector methods, interpreted as mathematical equations that define functions, always have a solution. This is required for soundness because in the BoogiePL program we introduce a function symbol for each inspector method, together with axioms that state that the functions satisfy the equations. If the system of equations has no solution, we obtain an inconsistent theory.

If the precondition of an inspector method  $m_1$  calls an inspector method  $m_2$  on **this**, then  $m_2$  must precede  $m_1$  textually in the program. The latter restriction is not necessary for soundness, but it ensures that if one were to recursively check inspector method preconditions at inspector method calls, including calls that occur inside inspector method preconditions, then such recursion would always terminate. It also means that the notion of recursive validity of an assertion with respect to a particular program state, defined as the assertion's validity condition being both recursively valid itself and true, is uniquely defined.

Due to ownership, it is no longer true for all method calls that if in the pre-state an allocated object is not in the callee's required write set, then that object is left unchanged by the call. Indeed, the object may be transitively owned by an object that is in the required write set. The call may then modify the object after performing the necessary unpack operations. A weaker property remains true, however: if in the pre-state of a call an object is in the caller's write set and is not in the callee's write set, then the call does not modify the object. This follows from the fact that being writable and being owned are mutually exclusive. We use this weaker property as a frame condition in the presence of ownership.

A consequence of the use of this new frame condition is that a method needs

to require writability of an object even if it only reads the object; the reason is that if the method performs method calls itself, it loses all information about objects that are not in its write set. A consequence of requiring writability of an object is that the method must specify the post-state of the object in its postcondition. If the method does not in fact modify the object, it must state equality of the post-state and the pre-state in some object-specific way, e.g. by stating  $o.getX() = \mathbf{old}(o.getX())$  for each inspector method  $getX$  of the object.

To avoid this inconvenience, we introduce a separate notion of *readability*. Specifically, we introduce a *read bag*  $R$  to complement the write set  $W$ . The read bag maps an object reference to a nonnegative integer. From the write set and the read bag, we derive the following: the *effective read set*  $R'$  is the set of all objects that in the write set or in the read bag. The *effective write set*  $W'$  is the set of all objects that are in the write set and not in the read bag. We also introduce *readability conditions*  $\mathbf{readable}(o)$ . The syntax  $\mathbf{writable}(o)$  translates into  $o \in W'$  and  $\mathbf{readable}(o)$  translates into  $o \in R'$ . Writing a field  $o.f$  is allowed only if  $o \in W'$ ; reading  $o.f$  or calling an inspector method on  $o$  is allowed only if  $o \in R'$ .

The above motivation for introducing readability conditions does not justify the restriction that a field  $o.f$  may be read only if  $o$  is effectively readable. Indeed, the system of this paper would be sound even if there were no restrictions on reading fields. However, we introduce the restriction because it is useful in some extensions of the system of this paper and because it is expected intuitively given the term *readability*. Also, it is probably true that at a point where a field  $o.f$  is read, one needs to prove that  $o$  is readable even without this restriction, in order to prove that  $o.f$  is not modified by method calls. Therefore, the restriction is unlikely to increase the annotation or proof burden significantly.

The above allows the following frame condition to be generated from a given method contract:

**free ensures**  $(\forall o \in \mathbf{old}(\text{dom } H) \bullet \mathbf{old}(o \in R' \wedge \neg P_W[o]) \Rightarrow H[o] = \mathbf{old}(H[o]));$

In other words, all objects that, in the pre-state, are effectively readable by the caller and are not in the required write set of the callee, are unchanged by the call.

In addition to supporting the generation of frame conditions, the distinction between writability and readability is useful for the verification of concurrent programs and quasi-concurrent programs, such as those that use coroutines, iterators, or callbacks. We do not discuss these further in this paper.

A method that has read-only access to an object may gain access to the object's owned objects through the new **read** block. The system under ownership

```

class IntList {
  rep int[] elems;
  int count;
  invariant  $0 \leq \textit{count} \wedge \textit{count} \leq \textit{elems.length}$ ;
  inspector int getCount() { return count; }
  inspector int getItem(int index)
    requires  $0 \leq \textit{index} \wedge \textit{index} < \textit{getCount}()$ ;
    { return elems[index]; }
  derived_invariant  $0 \leq \textit{getCount}()$ ;
  IntList(int[] xs)
    requires readable(xs);
    ensures writable(this)  $\wedge$  this.inv;
    ensures getCount() = xs.length;
    ensures forall{int i in (0 : getCount())}; getItem(i) = xs[i];
    { ... }
  void Add(int x)
    requires writable(this)  $\wedge$  this.inv;
    ensures writable(this)  $\wedge$  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;

```

Figure 7: A class that illustrates object invariants, ownership, parameterized inspector methods, and inspector method preconditions

is as follows.

$$R' = W \cup R \quad W' = W - R$$

$x := \mathbf{new} C; \equiv$ $x := \mathbf{new} C;$ $W[x] := \mathbf{true};$ $x.\mathit{inv} := \mathbf{false};$	$\mathbf{pack} o; \equiv$ $\mathbf{assert} o \in W';$ $\mathbf{assert} \neg o.\mathit{inv};$ $\mathbf{foreach} (p \in \mathit{rep}(o))$ $\quad \mathbf{assert} p \in W' \wedge p.\mathit{inv};$ $\mathbf{assert} \mathit{Inv}(o);$ $\mathbf{foreach} (p \in \mathit{rep}(o))$ $\quad W[p] := \mathbf{false};$ $\quad o.\mathit{inv} := \mathbf{true};$ $\mathbf{unpack} o; \equiv$ $\mathbf{assert} o \in W';$ $\mathbf{assert} o.\mathit{inv};$ $\mathbf{foreach} (p \in \mathit{rep}(o))$ $\quad W[p] := \mathbf{true};$ $\quad o.\mathit{inv} := \mathbf{false};$	$x := o.f; \equiv$ $\mathbf{assert} o \in R';$ $x := o.f;$ $o.f := x; \equiv$ $\mathbf{assert} o \in W'$ $\quad \wedge \neg o.\mathit{inv};$ $o.f := x;$
$\mathbf{read} (o) S \equiv$ $\mathbf{assert} o \in R' \wedge o.\mathit{inv};$ $R[o]++;$ $\mathbf{foreach} (p \in \mathit{rep}(o))$ $\quad R[p]++;$ $S$ $R[o]--;$ $\mathbf{foreach} (p \in \mathit{rep}(o))$ $\quad R[p]--;$		

## 5 Inheritance

### 5.1 Class Extension

We support inheritance by treating an object that is an instance of multiple classes as a composition of *frames*; for each object  $o$  and each class  $C$  of which  $o$  is an instance, we identify a frame  $(o, C)$ . The frame conceptually contains the fields of  $o$  declared in  $C$ . We introduce a separate *inv* bit in each frame. Also, the write set, read bag, effective write set, and effective read set are sets of frames rather than objects. Also, ownership is between frames. If an object  $o$  is an instance of a class  $D$  and  $D$  extends  $C$ , then frame  $(o, C)$  is a *rep frame* of  $(o, D)$ . Whenever  $((D)o).\mathit{inv}$ , frame  $(o, D)$  owns frame  $(o, C)$ . The **pack**, **unpack**, and **read** statements and the **writable** and **readable** expressions operate on frames. Object invariants and derived invariants apply only to the frame on which they are declared.

A **rep** field  $((C)o).f$  declared in a class  $C$  containing an object reference  $p$  contributes the frame  $(p, \mathit{classof}(p))$  to  $(o, C)$ 's set of rep frames.

An object expression used as the operand of a **writable** or **readable** expression or as the target of a read of the *inv* field is interpreted by default as the frame corresponding to the static type of the expression. To get the frame corresponding to the class of the object, wrap the object expression in a **dynamic** expression, e.g. **dynamic** $(o).\mathit{inv}$ .

### 5.2 Dynamic Binding

Methods, including inspector methods, may be virtual, and a class that transitively extends or implements a type  $T$  may override or implement non-final virtual methods, including inspector methods, declared by  $T$ . In our programming language, virtual method calls may be statically bound (using syntax  $o.m(\bar{a})$ ) or dynamically bound (using syntax **dynamic** $(o).m(\bar{a})$ ). Abstract methods (including interface methods) may not be called using a statically bound call.

A virtual method may declare two separate contracts: a static contract and a dynamic contract. The static contract is declared using the keywords **requires** and **ensures**. The dynamic contract is declared using the new **dynamic** keyword: **dynamic requires** and **dynamic ensures**. An abstract method (in-

cluding an interface method) may only have a dynamic contract. Consider a virtual method  $m$  in type  $T$ . For each method  $m'$  in a non-abstract class  $C$  that reflexively-transitively implements or overrides  $m$ , it must hold that the static contract of  $m'$ , if any, under the assumption that the class of the receiver is  $C$ , implies the dynamic contract of  $m$ . Note that it follows that a method body need only be verified against the method's static contract.

If a non-abstract method declares a dynamic contract but no static contract, as a convenience, the static contract is taken to be the dynamic contract with all occurrences of **dynamic(this)** replaced by **this**.

Non-virtual methods and constructors may not have dynamic contracts.

A dynamically-bound call is verified using the dynamic contract of the method to which the call is resolved at compile time. A statically-bound call, such as a **super** call, uses the static contract.

A confined expression confined to a frame  $(o, T)$ , where  $T$  is either a class  $C$  or the expression `classof(o)`, may only depend on state reflexively-transitively owned by  $(o, T)$ . For example, an expression confined to a frame  $(o, C)$  may not include dynamically-bound inspector method calls on  $o$ .

A type may declare *dynamic invariants*; these generate a proof obligation for each class that reflexively-transitively implements or extends the type. Specifically, if a class  $C$  declares or inherits a dynamic invariant **dynamic invariant**  $P$ ;, it must hold that  $C$ 's object invariant, under the assumption that the class of **this** is  $C$ , implies  $P$ .  $P$  may include inspector method calls.

### 5.3 Method Inheritance

In principle, we do not allow method inheritance. Method resolution and method binding do not look in superclasses. A class that directly or indirectly extends a class that declares a non-abstract virtual method must declare a method that overrides this method. It follows that a class with **final** methods is effectively **final** itself; it cannot be extended.

However, as a convenience, if the program text does not specify an override for a non-final method, a default override is generated automatically. Its static contract is the overridden method's dynamic contract, with **dynamic(this)** replaced with **this**, and its body unpacks the receiver, calls the overridden method, and packs the receiver.

Note, however, that default overrides are not necessarily correct; they generate proof obligations just like other methods do. For example, if a subclass declares an invariant, and a superclass method breaks this invariant, then the pack statement in the default override fails to verify, and an explicit override must be provided.

### 5.4 Example

Figure 8 shows a contrived example that illustrates many of these concepts. It declares a class *Cell* and a class *MyCell* that extends *Cell*. Class *Cell* declares a virtual inspector method *getX*. Class *Cell* backs this inspector method with a field  $x$  it declares. Class *MyCell*, however, implements the inspector by returning one less than the value of  $x$ .

Note: in a class  $D$  that extends a class  $C$ , the expression **super** is shorthand for  $((C)\mathbf{this})$ .

This example illustrates that our approach supports a complete separation between the two aspects that play a role when a class  $D$  extends a class  $C$ :

- $D$  implements  $C$ 's dynamic interface, which is used by polymorphic clients to access the most derived (i.e.  $D$ ) frame, and

```

class Cell {
  int x;
  invariant 0 ≤ x;
  inspector int getX() { return x; }
  derived_invariant 0 ≤ this.getX();
  dynamic invariant 0 ≤ dynamic(this).getX();
  Cell(int x)
    requires 0 ≤ x;
    ensures writable(this) ∧ this.inv;
    ensures this.getX() = x;
  { this.x := x; pack this; }
  void setX(int x)
    dynamic requires 0 ≤ x;
    dynamic requires writable(dynamic(this)) ∧ dynamic(this).inv;
    dynamic ensures writable(dynamic(this)) ∧ dynamic(this).inv;
    dynamic ensures dynamic(this).getX() = x;
    requires 0 ≤ x;
    requires writable(this) ∧ this.inv;
    ensures writable(this) ∧ this.inv;
    ensures this.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 writable(this) ∧ this.inv;
    ensures this.getX() = x;
  { super(x + 1); pack this; }
  void setX(int x)
    requires 0 ≤ x;
    requires writable(this) ∧ this.inv;
    ensures writable(this) ∧ this.inv;
    ensures this.getX() = x;
  { unpack this; super.setX(x + 1); pack this; }
}

```

Figure 8: An example of an inheritance hierarchy. Note: all methods are virtual and subclass methods override corresponding superclass methods.

- a  $D$  frame contains a  $C$  frame to help represent its internal state;  $D$  accesses the  $C$  frame using  $C$ 's static interface.

The dynamic interface of a class or interface consists of virtual inspector methods, dynamic invariants, and dynamic contracts of virtual methods. The static interface of a class consists of fields, the object invariant, derived invariants, non-abstract inspector methods, and static contracts of non-abstract methods.

## 6 Discussion

**Run-time checking** The full methodology can be checked statically (i.e. verification conditions can be generated, which an automatic theorem prover may or may not be able to prove or refute), or alternatively, at run time. The advantage of the former is that it verifies all possible executions. The advantage of the latter is that it requires no method contracts. However, conformance is checked with respect to those method contracts that are provided. There are two modes of run-time checking: one where the read bag and write set that are maintained are the program's read bag and write set, and one where the read bag and write set are the method's read bag and write set. In the latter case, a stack of read bags and write sets is maintained. At each call, a new element is pushed onto the stack based on the accessibility (i.e. readability and writability) conditions in the method contract. This means that this approach enforces frame conditions at run time. This run-time checking involves a check at each write operation. This can be optimized by performing a static analysis to combine multiple checks into one.

### 6.1 Related Work

Müller [11] combines abstraction functions with an ownership type system called *Universes*. This type system is more restrictive than the Boogie ownership system which we use.

Leino and Müller [10] achieve state abstraction by combining the Boogie ownership system with a notion of model fields. Model fields in [10] 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 implements 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 [10]. 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, [10] 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 8.)

To support the kind of specifications enabled in our approach by parameterized inspector methods, such as the *getItem* method in the *IntList* example, [10] would have to use model fields containing special-purpose immutable objects such as immutable list objects (known as *model types* in JML [9]).

Kassios [8] 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 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, and it does not deal with the issue of validity of specifications, since in the framework used all functions and operators are total. Finally, the proposed approach has not been applied in the context of an automatic program verifier.

Darvas and Müller [5] 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 its 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 receiver object 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 exploiting the ownership relation, as suggested in [5]. Specifically, we allow an inspector method body to call an inspector method only on an owned object, which ensures that inspector method calls always terminate (since the ownership graph is finite and acyclic) 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.

Our write sets and read bags are similar to type systems that combine ownership and effects [4, 3].

An interesting development is the use of separation logic for verification of object-oriented programs with state abstraction [13, 12].

## 7 Conclusion

We presented a sound approach to the modular static verification of object-oriented programs that combines the Boogie ownership system and object invariant methodology with inspector methods to achieve state abstraction. Inspector

methods may have parameters. Derived invariants and dynamic invariants may be used to conveniently inform clients of properties of and relationships between inspector methods. Inheritance is fully supported, with full separation between the two aspects of subclassing, i.e. interface re-implementation and containment. Frame conditions are derived from writability conditions in preconditions and do not impose separate proof obligations.

## Acknowledgements

The authors thank K. Rustan M. 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] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004.
- [4] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proc. OOPSLA*, 2002.
- [5] m Darvas and Peter Muller. 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.
- [6] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.
- [7] Manuel Fahndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, 2003.
- [8] 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.
- [9] 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.
- [10] K. Rustan M. Leino and Peter Muller. A verification methodology for model fields. In *Proc. ESOP*, 2006. To appear.
- [11] Peter Muller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
- [12] Matthew J. Parkinson. *Local reasoning for Java*. PhD thesis, Computer Laboratory, Cambridge University, 2005.
- [13] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proc. POPL*, 2005.