

# Inferring Object Invariants

(Extended Abstract)

Bor-Yuh Evan Chang<sup>0</sup>

*University of California, Berkeley, CA, U.S.A.*

K. Rustan M. Leino<sup>1</sup>

*Microsoft Research, Redmond, WA, U.S.A.*

---

## Abstract

The program state for object-oriented languages, such as Java or C#, consists of both variables local to procedures and variables stored in the global heap. The variables stored in the heap are the fields of objects (*i.e.*, fields of class instances). This paper proposes a technique for inferring properties of such object-oriented programs. The technique divides the work into two interacting parts: a flow-sensitive analysis that tracks the local and global state as seen at each particular program point, and a flow-insensitive analysis that tracks properties that are common among all “valid” instances of a class, independent of the program counter. The analysis is sound and works in the presence of many object references (aliasing). For precision, the analysis makes use of a recent methodology for object invariants, which makes explicit when each object’s invariant is supposed to hold (valid objects) or is allowed to be violated.

*Key words:* object invariants, abstract interpretation,  
object-oriented programs, inference

---

## 0 Introduction

Proving the correctness of a computer program requires knowledge about the program that is not readily available in the program text. For example, to prove the correctness of a loop, one may need to know that a condition, such as  $0 \leq x$ , is a loop invariant. Or, to prove the correctness of a class in an object-oriented program, one may need to know that a condition  $0 \leq \mathbf{this}.f$  is invariant for all instances (objects) of a class. Such a condition is called an

---

<sup>0</sup> E-mail: bec@cs.berkeley.edu

<sup>1</sup> E-mail: leino@microsoft.com

```

class A {
  int x;
  A() {
    this.x := 8;
  }
  method M() {
    this.x := this.x + 1;
  }
}

class B {
  int x;
  B() {
    this.x := 8;
  }
  method M(B b) {
    this.x := b.x + 1;
  }
}

class C {
  int x;
  C() {
    this.x := 8;
  }
  method M(C c) {
    c.x := 7;
  }
}

```

Fig. 0. Simple program examples that demonstrate the need to consider object references.

*object invariant.* Since we cannot always rely on the program text to include declarations of the needed invariants, we consider in this paper the technique of *abstract interpretation* [6] to automatically infer such invariants from the text of an object-oriented program.

### Object references

Let us use some examples to home in on what our program analysis needs to deal with. Consider the Java-like classes in Figure 0. For class  $A$ , the condition  $\mathbf{this}.x \geq 8$  holds on exit from the constructor and on entry to and exit from the method  $M$ , and thus, we wish to obtain from our analysis that  $\mathbf{this}.x \geq 8$  is an object invariant for  $A$ . Since conditions like  $x \geq 8$  can be inferred using, for example, the polyhedra domain [7], a straw-man proposal for inferring the object invariant  $\mathbf{this}.x \geq 8$  is to treat the field select expression  $\mathbf{this}.x$  as denoting one variable, essentially focusing the analysis on one (arbitrary) object at a time. Then, perform abstract interpretation on the body of constructors and methods (*e.g.*, in the class setting of Logozzo [13]) to infer the condition  $\mathbf{this}.x \geq 8$ , from which one would conclude  $(\forall a: A \bullet a.x \geq 8)$ .

But this straw man has problems. Class  $B$  illustrates the need to know about properties of objects other than  $\mathbf{this}$ . Since it focuses on what can be said about  $\mathbf{this}.x$ , the straw man gives no information about  $b.x$  in method  $B.M$ , and thus we are not be able to conclude the fact  $(\forall b: B \bullet b.x \geq 8)$ . Perhaps more critically, class  $C$  demonstrates that the straw man is unsound. Method  $C.M$  updates the  $x$  field of an object other than  $\mathbf{this}$ , so the straw man might not consider this update as having any effect on the object invariant of  $C$  objects. The analysis would then incorrectly infer  $\mathbf{this}.x = 8$ , which would suggest the fallacious  $(\forall c: C \bullet c.x = 8)$ . We conclude from classes  $B$  and  $C$  that our analysis must know about object references.

To handle object references, we build on the abstract-interpretation framework for heap structures in our previous work [5]. In this framework, we treat a field read expression  $o.x$  in the source language as indexing into an explicit heap (represented by the variable  $H$ ) at location  $\langle o, x \rangle$ , where  $o$  is the object reference (*i.e.*, object identity, pointer, or address) and  $x$  is a uniquely valued

constant that names the field  $x$ , written as  $\text{sel}(H, o, x)$  (cf. [9,11,17]). To handle field updates, our framework additionally uses a *heap succession* predicate  $H \equiv_{o,x} H'$ , which asserts that heaps  $H$  and  $H'$  store the same values for all locations except possibly for the location  $\langle o, x \rangle$ .

Starting from an initial state represented by an abstract-domain element  $L$ , performing a forward analysis on a field update statement  $o.x := e$  amounts to applying the following abstract-domain operations, where  $H'$  is a fresh variable:

$$\begin{aligned} L &:= \text{Constrain}(L, \text{sel}(H', o, x) = e) ; \\ L &:= \text{Constrain}(L, H \equiv_{o,x} H') ; \\ L &:= \text{Eliminate}(L, H) ; \\ L &:= \text{Rename}(L, H', H) ; \end{aligned}$$

These operations introduce a name  $H'$  for the heap after the update statement, and constrain  $H'$  using  $\text{sel}$  and the heap succession predicate. Then, the old heap  $H$  is projected out and the name heap  $H'$  is renamed to  $H$ .

Most of the detailed workings of our abstract-interpretation framework are not relevant to our present paper. However, one feature that is important is that the framework is parameterized by various “off-the-shelf” abstract domains (*i.e.*, ones that reason about relations on variables as independent coordinates). For example, parameterizing the framework with the polyhedra domain [7], we obtain an analysis that can reason about references into a heap and linear arithmetic. Such an instantiation of the framework is thus able to infer that the statements  $o.x := 5 ; p.y := o.x + 12$  result in a state where  $o.x = 5 \wedge p.y = 17$  holds.

It is also worth pointing out that our framework correctly handles the *aliasing issues* that arise when distinct variables reference the same object. In the example we just gave, aliasing is not an issue (that is, it doesn’t matter if  $o$  and  $p$  refer to the same object), because the example updates two different fields,  $x$  and  $y$ . For an example like  $\dots ; o.x := 5 ; p.x := o.x + 12$ , the analysis (rightly) infers  $o.x = 5$  to hold in the final state only if there is information that implies that  $o$  and  $p$  do not reference the same object.

## A methodology for object invariants

Note that the problems with the classes  $B$  and  $C$  above arise even if there are no aliasing issues. What the examples show is that our analysis must somehow simultaneously keep track of the state of multiple objects (like  $\text{this}.x$ ,  $b.x$  in  $B.M$ , and  $c.x$  in  $C.M$ ). This situation may look pretty grim, because there is no bound on the number of different objects and all of these objects can simultaneously be in different states. As a solution, we abstract over a particular subset of objects and keep track of their states collectively.

We say that each object is in one of two states: it is either *valid*, which means its object invariant can be relied upon, or it is *mutable*, which means its object invariant may temporarily be violated. It is the set of valid objects that our analysis keeps track of collectively. The valid objects are class instances

in their “steady states”, which means it’s plausible to imagine that these objects satisfy some common constraint, namely the *object invariant* of their respective classes.

We make the following assumptions about the object-oriented code we analyze:

- an object  $o$  transitions from mutable to valid by the execution of a special statement **pack**  $o$
- an object  $o$  transitions from valid to mutable by the execution of a special statement **unpack**  $o$
- a field update  $o.x := e$  takes place only when the object  $o$  is mutable

These assumptions were introduced in the Boogie methodology for object invariants [0,12,2], which is used in the object-oriented Spec# programming language [1]. Note that the **pack** and **unpack** statements need not be provided directly by the programmer; they could be implied by rules in the source language or by rules in some stylized use of the language (*e.g.*, **unpack** on method entry and **pack** on method exit).

The Boogie methodology also treats class inheritance. It does so by having a mutable/valid state bit for each *class frame* of an object. A class frame is the set of declared fields of a particular class. For example, if  $B$  is a subclass of  $A$ , then a  $B$  object can be valid for  $A$  and mutable for  $B$ . This allows each subclass to declare its own object invariant. The statements we consider are therefore **pack**  $o$  **as**  $T$  and **unpack**  $o$  **from**  $T$ , which change object  $o$ ’s mutable/valid bit for class frame  $T$ . If  $x$  is a field declared in a class  $T$ , then we assume a field update  $o.x := e$  takes place only when  $o$  is mutable for  $T$ .

## Overview of our technique

The technique for inferring object invariants that we propose in this paper performs two analyses that work together. One analysis, the *local analysis*, proceeds flow-sensitively and infers properties of the program at each program point, which is a standard thing to do in abstract interpretation. This analysis more or less determines properties of local variables and heap locations reachable from the local variables. The other analysis, the *global analysis*, proceeds flow-insensitively and infers, for each class  $T$ , properties of all valid  $T$  objects—that is, the object invariant for class  $T$ .

When the local analysis encounters a valid object of some type  $T$ , the global analysis’ current approximation of the object invariant for  $T$  objects is instantiated to give the local analysis more information. When the local analysis reaches a program point where the invariant of a given object is supposed to hold, the information it has about that object is used by the global analysis for its next approximation.

## 1 Inferring Object Invariants

In this section, we describe our technique to infer object invariants using the abstract interpretation-based framework discussed in the previous section. Recall that the framework is parameterized by various traditional abstract domains, which determine the kind of invariants that we infer about objects. To simplify the discussion, we consider a single such abstract domain, which we shall call the *policy abstract domain*. In other words, we describe a flexible mechanism by which object invariants can be inferred and defer to the user the choice of policy that determines the kind of invariants that are inferred (as well as, how precise the analysis is).

Let  $\mathcal{P}$  denote the given policy domain. As alluded to earlier, we can transparently extend the policy domain to infer properties about heap structures using the framework described in our previous work [5]. We denote the abstract domain given by this extension by  $\mathcal{C}(\mathcal{P}, \mathcal{S})$  where  $\mathcal{S}$  indicates the presence of the abstract domain that handles the heap succession predicate for heap updates.

The abstract state of the local flow-sensitive analysis is precisely an element of  $\mathcal{C}(\mathcal{P}, \mathcal{S})$ . Meanwhile, the global flow-insensitive analysis tracks the object invariant for each class  $T$ , which is simply a constraint on fields of *valid* objects of class  $T$  representable by  $\mathcal{C}(\mathcal{P})$ , the policy domain extended to reason on heap locations. Thus, the abstract state of the global analysis is a mapping from classes to elements of  $\mathcal{C}(\mathcal{P})$ , which we denote by  $\mathcal{J}$ . (The heap succession domain  $\mathcal{S}$  is not needed to represent object invariants, as it is only needed to reason about updates to heaps whereas the object invariant applies globally to all heaps.) More precisely, the concretization of an element of  $\mathcal{J}$  is a conjunction of universally-quantified propositions of the following form:

$$(\forall H \bullet (\forall t: T \bullet \text{Valid}(t) \Rightarrow \text{ToPredicate}_{\mathcal{C}(\mathcal{P})}(C)))$$

where  $C$  is an element of  $\mathcal{C}(\mathcal{P})$  whose only free variables are  $t$  and  $H$ . Moreover, the `sel` expressions in  $\text{ToPredicate}_{\mathcal{C}(\mathcal{P})}(C)$  should be of the form `sel( $H, t, x$ )` for any field  $x$  of  $T$  or superclass thereof. (In the literature [6], the concrete domain is often described as sets of program states rather than as state predicates and `ToPredicate` is written as  $\gamma$ .)

We now describe the abstract interpretation by defining the following abstract transition judgment for a statement  $s$ :

$$s: \langle I \ ; C \rangle \rightarrow \langle I' \ ; C' \rangle$$

where  $I$  and  $I'$  are elements of  $\mathcal{J}$  and  $C$  and  $C'$  are elements of  $\mathcal{C}(\mathcal{P}, \mathcal{S})$ . We write the combined abstract state  $\langle I \ ; C \rangle$  at each program point for notational convenience, but while there is one  $C$  per program point as in standard abstract interpretation, there is only one global  $I$ . As usual, the  $C$  is initially bottom ( $\perp_{\mathcal{C}(\mathcal{P}, \mathcal{S})}$ ) at each program point except in some initial state where it

is top ( $\top_{\mathcal{C}(\mathcal{P}, \mathcal{S})}$ ). Similarly, the global  $I$  is also initially bottom (*i.e.*, each class  $T$  is mapped to  $\perp_{\mathcal{C}(\mathcal{P})}$ ).

The global and local components of the abstract state interact precisely at the **unpack** and **pack** statements. At each statement **pack**  $o$  **as**  $T$ , the program asserts the object invariant for class  $T$  to hold for the object  $o$ , so we incorporate any properties inferred about the fields of  $o$  into the local analysis. Roughly speaking, we need to obtain from  $C$  the constraints involving fields of  $o$  declared in  $T$  or any superclass of  $T$  (*i.e.*, expressions of the form  $\text{sel}(H, o, x)$  for fields  $x$  of class  $T$ ). Then, rename  $o$  to  $t$  in the local invariant and weaken the current global  $I$  with it. As we show with an example below, this weakening must be performed with a widen operator to ensure termination of the analysis, not simply a join. (As usual [6], one can use some sequence of successively coarser widening operators of which the first few may be join operations.) More precisely, the handling of **pack** is as follows:

$$\frac{P = C \uparrow \text{sel}(H, o, *T)}{\text{pack } o \text{ as } T: \langle I \ ; C \rangle \rightarrow \langle I[T \mapsto I(T) \nabla [t/o]P] \ ; C \rangle}$$

where  $H$  is heap variable for the current heap. We write  $\nabla$  for the widen operator and  $[y/x]e$  for the capture-avoiding renaming of variable  $x$  to  $y$  in expression  $e$ . Also, we write  $C \uparrow e$  for the operation that may drop constraints from  $C$ , keeping only those constraints that have no free variables other than those occurring in subexpressions that match  $e$  (in which “ $*T$ ” matches any field declared in class  $T$  or superclass of  $T$ ). As applied above, this operation obtains the constraints that involve only the fields of  $o$ . For example, if  $x, y, z$  represent fields declared in class  $T$  and  $C$  represents  $\text{sel}(H, o, x) \leq 5 \wedge \text{sel}(H, o, y) \leq \text{sel}(H, p, z)$ , then what  $C \uparrow \text{sel}(H, o, *T)$  returns represents  $\text{sel}(H, o, x) \leq 5$ .<sup>0</sup>

One may observe that  $C$  is an element of  $\mathcal{C}(\mathcal{P}, \mathcal{S})$ , while  $I(T)$  is an element of  $\mathcal{C}(\mathcal{P})$ , so the above rule is not quite well-formed. To communicate between the two analyses, the language of discourse is that of first-order logic—we first concretize and extract the predicate of interest from  $C$  and then abstract in the abstract domain  $\mathcal{C}(\mathcal{P})$  before widening. One can view  $C \uparrow e$  as a special filtering  $\text{ToPredicate}_{\mathcal{C}(\mathcal{P}, \mathcal{S})}$ .

During the local analysis, we may need to instantiate an object invariant to obtain a certain property at a particular point. The object invariant methodology ensures that changes to fields are guarded by an **unpack** statement and thus indicates when the object invariant may be temporarily violated. We instantiate object invariants at **unpack** statements, as it marks the code section that weakens the object invariant. More precisely, an **unpack** statement

<sup>0</sup> The implementation of the  $\uparrow$  operation uses the fact that our framework maps all subexpressions to symbolic values [5], but we lack the space to discuss those details here.

proceeds as follows:

$$\overline{\mathbf{unpack} \ o \ \mathbf{from} \ T: \langle I \ ; C \rangle \rightarrow \langle I \ ; C \sqcap [o/t]I(T) \rangle}$$

Similar to **pack**, the actual interaction is via a conversion of  $I(T)$  to a predicate (concretization) and then an abstraction in the abstract domain  $\mathcal{C}(\mathcal{P}, \mathcal{S})$  before the meet ( $\sqcap$ ) with  $C$ .

All other program statements affect only the local abstract state  $C$  and proceed basically as standard abstract interpretation (except the reasoning is extended to work with heap structures [5]). Note that this separation is only obtained because of the object invariant methodology described in Section 0. In particular, a field update has no affect on the global abstract state  $I$  because the methodology dictates that only a mutable (*i.e.*, unpacked) object can be modified—recall that the object invariant carried in  $I$  applies only to valid (*i.e.*, packed) objects.

### Example

Figure 1 gives an example analysis of a slightly modified version of class  $B$  from Figure 0 with explicit **unpack** and **pack** statements. We assume the polyhedra domain [7] is used as the policy domain. The abstract state at each program point is shown right-justified and boxed. For this example, we assume that we have as a precondition that  $d$  is valid, which might be user-specified, implied by the rules in the source language or some stylized use of the language, obtained by a separate analysis, or obtained simultaneously with this analysis. In either case, we have in the initial state of method  $M$  that  $\mathbf{sel}(H, d, y) = 1 \wedge \mathbf{sel}(H, d, x) = 8$  through an instantiation of the current object invariant for  $D$  and  $B$ . The **unpack** statement in  $M$  instantiates the current object invariant of  $B$  for **this**. The field update statement does not affect the global state and proceeds as usual on the local state. Then, at the **pack** statement in  $M$ , we get from the local state that  $\mathbf{sel}(H, \mathbf{this}, x) = 9$ . This state is widened into the current object invariant for  $B$  to get that  $\mathbf{sel}(H, t, x) \geq 8$  for the new object invariant for  $B$ . Since this widening brought about a weakening of the current approximation of the object invariant for  $B$ , the analysis needs to revisit all program points where the previous approximation for  $B$  was instantiated, namely at those points marked with an  $*$  in Figure 1. Continuing the analysis with the weaker object invariants then reaches a fixed point.

The example in Figure 1 demonstrates why the weakening of the object invariant must be with a widen operator, just as in standard abstract interpretation with looping constructs. In the example, using joins to weaken the object invariant would yield the infinite ascending chain of abstract domain elements

$$9 \geq \mathbf{sel}(H, t, x) \geq 8 \quad 10 \geq \mathbf{sel}(H, t, x) \geq 8 \quad 11 \geq \mathbf{sel}(H, t, x) \geq 8 \quad \dots$$

```

class D extends B {
  int y ;
  D() {  $\langle D \mapsto \perp \S \top \rangle$ 
    this.y := 1 ;  $\langle D \mapsto \perp \S \text{sel}(H, \mathbf{this}, y) = 1 \rangle$ 
    pack this as D ;  $\langle D \mapsto \text{sel}(H, t, y) = 1 \S \text{sel}(H, \mathbf{this}, y) = 1 \rangle$ 
  }
}

class B {
  int x ;
  B() {  $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \perp \S \top \rangle$ 
    this.x := 8 ;  $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \perp \S \text{sel}(H, \mathbf{this}, x) = 8 \rangle$ 
    pack this as B ;  $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \text{sel}(H, t, x) = 8 \S \text{sel}(H, \mathbf{this}, x) = 8 \rangle$ 
  }
  method M(D d) {
     $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \text{sel}(H, t, x) = 8 \S \text{sel}(H, d, y) = 1 \wedge \text{sel}(H, d, x) = 8 \rangle$  *
    unpack this from B ;
     $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \text{sel}(H, t, x) = 8 \S \text{sel}(H, d, y) = 1 \wedge \text{sel}(H, d, x) = 8 \wedge \text{sel}(H, \mathbf{this}, x) = 8 \rangle$  *
    this.x := d.x + d.y ;  $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \text{sel}(H, t, x) = 8 \S \text{sel}(H, \mathbf{this}, x) = 9 \rangle$ 
    pack this as B ;  $\langle D \mapsto \text{sel}(H, t, y) = 1, B \mapsto \text{sel}(H, t, x) \geq 8 \S \text{sel}(H, \mathbf{this}, x) = 9 \rangle$ 
  }
}
    
```

Fig. 1. An example analysis.

We note that the class argument to **unpack** and **pack** need not be the surrounding class as it is in this example. For example, specifying a superclass would instantiate and update the object invariant of the superclass.

### Object invariants that mention more than one object

The technique for inferring object invariants we have described thus far does not consider object invariants involving multi-level field accesses, which often are quite important. For example, we may implement a *Set* class with an array as the backing store where an object invariant is that the cardinality of the set is equal to length of the array (see inset). Without additional support, soundly inferring such an

```

class Set {
  int card ; int[] arr ;
  invariant this.card = this.arr.Length ;
  ...
}
    
```

invariant is hard because of aliasing issues—an alias of the *arr* field of a *Set* object *o* could modify the array to which *arr* points without going through *o.arr*. Fortunately, the object invariant methodology [0] also helps us with this problem: an ownership model is imposed, and such an object invariant is allowed only if *arr* is declared to be “owned” by *Set* (*i.e.*, declared a **rep** field). The details of this ownership model are not relevant here; what is necessary is that the methodology assures us that *arr* cannot be made mutable unless *o* is first made mutable (*i.e.*, unpacked). This allows us to instantiate the object invariant for any owned field (*e.g.*, *arr*) of an object *o* when *o* is unpacked and incorporate any information about the fields of owned fields (*e.g.*, *arr*) when *o* is packed (and transitively for the owned fields of the owned fields of *o*). In the context of the analysis described in Section 1, the  $\cdot \uparrow \cdot$  operation that pulls out constraints involving the fields is extended to also get fields of owned fields (and transitively). The condition above the line in the rule for **pack** then takes a form like (here shown without transitivity):

$$P = C \uparrow \{ \text{sel}(H, o, *_T), \text{sel}(H, (\text{sel}(H, o, *\text{rep}_T)), *) \}$$

where  $*_{\text{rep}_T}$  matches any **rep** field declared in *T* or a superclass of *T*, and  $*$  matches any field.

## 2 Related Work

Inferring properties of object-oriented programs poses several challenges. One challenge is the fact that objects can be aliased, that is, distinct variables may refer to the same object. Various pointer analyses exist to address such problems (see, for example, [19,8]). These pointer analyses can then be used as a basis for generating abstract locations on which traditional abstract interpretation can be applied [3,20]. Another challenge is determining the structure of pointers in the object heap—a technique that goes under the name of *shape analysis* [16], which has also been used as a basis for abstract interpretation [18]. Another challenge is analyzing properties of object fields [5], or considering combinations of the methods in a class [13]. The challenge we address in this paper is inferring relations among the fields of an object, and in particular inferring the field relations (object invariants) that are the same for all instances of a class.

Our technique uses a combination of a flow-sensitive analysis and a flow-insensitive analysis. Venet and Brat have also used a combination of such analyses [20], though their flow-insensitive analysis is used instead to refine set of abstract locations obtained from an initial points-to analysis. The analogous reasoning is incorporated in our flow-sensitive analysis via the  $\mathcal{C}(\cdot, \mathcal{S})$  framework. Logozzo [14] has also considered the analysis of object-oriented languages in the presence of aliasing and a *context* of other objects. In his recent PhD thesis, Lahiri [10] has considered inferring quantifications whose bodies are boolean combinations of predicates  $P(k)$ , where  $k$  is the quantified

variable and  $P$  comes from some fixed set of predicate symbols. His technique also performs a join operation to weaken the body of the quantifier by new facts inferred in a flow-sensitive manner.

### 3 Conclusion

Summarizing our contributions, we have presented a technique for inferring object invariants in object-oriented programs. Aimed at mainstream object-oriented languages, the technique handles aliasing among objects and works in the presence of multiple objects that can be in different states. Our technique splits the work into two collaborating parts: a flow-sensitive analysis that infers invariants for each program point and a flow-insensitive analysis that infers invariants of all valid objects of a class. Building on our previous work on combining abstract domains and handling heap structures [5], our technique can be parameterized by arbitrary abstract domains that determine what kinds of invariants are inferred. To determine in a structured way when objects are in “steady states”, we use the Boogie object-invariant methodology [0].

To our knowledge, our technique is the first to, in the presence of multiple objects, infer object invariants that hold for all objects of a class.

The technique itself is one of the first to divide up the work into a combination of a flow-sensitive part and a flow-insensitive part. A possibly surprising aspect of our work is that we do not apply a global alias analysis. Instead, we use information available locally to resolve aliasing issues. This makes our technique more modular, and we expect it to work well in settings where programmers have the ability to write preconditions of methods (like in Eiffel [15], Java with JML [4], and Spec# [1]). A final contribution of our work is to combine a methodology and an analysis, which adds power to both. The methodology gives the analysis guidance about the form of what is to be inferred and where to infer it, and the analysis makes the methodology easier to use by supplementing any explicit invariants the programmer might have declared. In fact, we posit that because of the methodology, we could get quite precise invariants even with a cheap conservative intraprocedural analysis where a call is modeled as basically scrambling the entire heap, for we can assume after the call that the object invariant of any object that is not passed to the callee in the mutable (*i.e.*, unpacked) state holds.

We are implementing our technique in the Spec# [1] program verifier and are looking forward to getting some experience with it.

### References

- [0] Barnett, M., R. DeLine, M. Fähndrich, K. R. M. Leino and W. Schulte, *Verification of object-oriented programs with invariants*, Journal of Object Technology **3** (2004), pp. 27–56.

- [1] Barnett, M., K. R. M. Leino and W. Schulte, *The Spec# programming system: An overview*, in: *CASSIS 2004*, LNCS **3362** (2005), pp. 49–69.
- [2] Barnett, M. and D. A. Naumann, *Friends need a bit more: Maintaining invariants over shared state*, in: *Mathematics of Program Construction 2004*, LNCS **3125** (2004), pp. 54–84.
- [3] Blanchet, B., P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *A static analyzer for large safety-critical software*, in: *PLDI 2003* (2003), pp. 196–207.
- [4] Burdy, L., Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll, *An overview of JML tools and applications*, International Journal on Software Tools for Technology Transfer (2005).
- [5] Chang, B.-Y. E. and K. R. M. Leino, *Abstract interpretation with alien expressions and heap structures*, in: *VMCAI 2005*, LNCS **3385** (2005), pp. 147–163.
- [6] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Fourth POPL*, 1977, pp. 238–252.
- [7] Cousot, P. and N. Halbwachs, *Automatic discovery of linear restraints among variables of a program*, in: *Fifth POPL*, 1978, pp. 84–96.
- [8] Das, M., *Unification-based pointer analysis with directional assignments*, in: *PLDI 2000* (2000), pp. 35–46.
- [9] Hoare, C. A. R. and N. Wirth, *An axiomatic definition of the programming language PASCAL*, Acta Informatica **2** (1973), pp. 335–355.
- [10] Lahiri, S. K., “Efficient techniques for Unbounded System Verification,” Ph.D. thesis, Carnegie Mellon University (2004), to appear.
- [11] Leino, K. R. M., “Toward Reliable Modular Programs,” Ph.D. thesis, California Institute of Technology (1995), Technical Report Caltech-CS-TR-95-03.
- [12] Leino, K. R. M. and P. Müller, *Object invariants in dynamic contexts*, in: *ECOOP 2004*, LNCS **3086** (2004), pp. 491–516.
- [13] Logozzo, F., *Class-level modular analysis for object oriented languages*, in: *SAS 2003*, LNCS **2694** (2003), pp. 37–54.
- [14] Logozzo, F., *Separate compositional analysis of class-based object-oriented languages*, in: *AMAST 2004*, LNCS **3116** (2004), pp. 332–346.
- [15] Meyer, B., “Object-oriented Software Construction,” Series in Computer Science, Prentice-Hall International, 1988.
- [16] Muchnick, S. S. and N. D. Jones, *Flow analysis and optimization of Lisp-like structures*, in: S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981 pp. 102–131.
- [17] Poetzsch-Heffter, A., *Specification and verification of object-oriented programs*, Habilitationsschrift, Technische Universität München (1997).
- [18] Sagiv, M., T. W. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, TOPLAS **24** (2002), pp. 217–298.
- [19] Steensgaard, B., *Points-to analysis in almost linear time*, in: *23rd POPL* (1996), pp. 32–41.

- [20] Venet, A. and G. Brat, *Precise and efficient static array bound checking for large embedded C programs*, in: *PLDI 2004* (2004), pp. 231–242.