

Verification of Object-Oriented Programs with Invariants

Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte

Microsoft Research, Redmond, WA, USA

{mbarnett,rdeline,maf,leino,schulte}@microsoft.com

Manuscript KRML 122a, 2 May 2003.

Abstract. This extended abstract outlines a system for the modular verification of an object-oriented programming language. While simplified, the language has object and array references, single-inheritance subclassing, and single-dispatch methods. Programs are verified against specifications consisting of *pre- and post-conditions* for methods, and *object invariants* stating the consistency of data. The meaning of a program is given by its translation into *verification conditions*: logical formulas that are valid if and only if the program is consistent with its specification. The technique is *sound* and is expressive enough to allow many interesting object-oriented programs to be specified and verified. The key idea is to keep track of, for each object, which invariants the object satisfies and whether or not the object can be mutated.

0 Introduction

Writing correct programs is difficult, and useful modern programming-language features like information hiding and object orientation provide several challenges in reasoning about programs. In this extended abstract, we consider a programming methodology based on method specifications and object invariants and define the correctness criteria for programs specified in that way. The key idea is to keep track of which object invariants each object satisfies at each program point (which we do by introducing for each object a special field *inv*) and whether or not the object is in a state where it can be mutated (which we do by introducing for each object a special field *exposed*).

The main characteristics of this work are:

- *Ease of use.* Our verification system needs only a relatively small repertoire of programming concepts. This makes it simpler than previous verification approaches. For instance, alias confinement and readonly modifiers are not first-class concepts in our methodology. Also, our technique does not require as much foresight on behalf of the program designers as does the *valid/state paradigm* in ESC/Modula-3 [4, 9]. Perhaps surprisingly, our technique does not even use features of abstraction to specify which variables a method is allowed to modify (like abstract variables and abstraction dependencies [9] or data groups [10]).

- *Simple foundation.* The meaning of programs is given by a translation from programs to *verification conditions*, thus following the approach ESC/Modula-3. Verification conditions are expressed in first-order predicate calculus. Thus, we do not need any special logics, like separation logics (*e.g.*, [0]), or coalgebras as used in the LOOP project [8], to reason about pointer structures and updates. This makes our approach more amenable to further studies on heap structures and confinement.
- *Modularity.* The proposed technique is modular in that one can verify software units (say, methods, classes, or small sets of classes) separately and independently from the rest of the program. In contrast to whole-program analysis, this supports scalability and fast response times.
- *Soundness.* We claim our technique to be *modularly sound*: if the technique detects no violations in the independently verified software units, then there are no violations in the program as a whole. Modular soundness was a goal in ESC/Modula-3, but it was achieved only for the most basic forms of its specification language. Using a different encoding of similar specifications, Müller and Poetzsch-Heffter’s technique [13, 12] achieves modular soundness.

Other related static-checking systems are Vault [2, 6] and Fugue [3], which incorporate some modern alias confinement techniques like *capabilities* [16]. Our approach has a similar methodology, but aims at analyzing the programs at a more detailed level.

1 Methodology

As a first part of our programming methodology, each object can either be *exposed* or not. A program can only write the fields on an object when it is exposed. When an object is not exposed, it exists as an integral *component* of some other object and is said to be *owned* by that other object.

Although a program cannot write the fields of an owned object, the program can read its fields. The idea is that an owned object is encapsulated; other objects should not care or depend upon its details and should not be surprised if its details change as the result of any method call.

In an object-oriented language, an object can have multiple declared types due to inheritance. We allow the programmer to specify an invariant in each class declaration, which means that an object’s complete invariant consists of the invariants of all the inherited classes. As a second part of our methodology, we allow a method to specify which subset of an object’s total invariant holds in the method’s pre- and postcondition. For simplicity, this extended abstract considers an object-oriented language with single inheritance, like C# or Java, but without the **interface** feature of these languages. As part of our methodology, we restrict the invariant subset to be a *prefix* of an object’s invariants, from those declared in the root class **object** toward those declared in the allocated type of the object. We represent this invariant prefix using a single type, namely, the most derived type whose invariant holds.

The two parts of the methodology are related by the following two properties:

0. If, for an object, the invariants declared in a class T are known to hold, then all of the *components* of the object declared in class T are *owned* by the object.
1. If an object is owned, then all its object invariants are known to hold.

We now make these notions more precise. Every object is extended by two instance fields, *exposed* and *inv*. The field *exposed* holds a boolean that represents whether the object's fields can be written. The field *inv* holds a type that records the most derived class whose declared invariants are known to hold for the object. If no invariants are known to hold for the object, *inv* holds the special value \perp .

The fields *exposed* and *inv* are special in that they can be mentioned in method specifications but not directly read or written in method bodies. To update these special fields, we introduce two statements **pack** and **unpack**, which delineate the scope in which invariants hold. The statement **unpack** *o* **from** *T* changes *o.inv* from *T* to the superclass of *T*. Statements after **unpack** can update fields and sub-components declared in *T* and violate *T*'s declared invariant. The statement **pack** *o* **as** *T* checks that *T*'s declared invariant holds and changes *o.inv* from the superclass of *T* back to *T*. This description should be sufficient to study a small example, in which the meaning of **pack** and **unpack** is explained in greater detail.

2 Example

We consider a class *Bag*, whose instances represent multisets (bags) of integers. The implementation has two fields: an integer *n* that stores the cardinality of the bag, and an array *a* whose first *n* elements are the elements of the bag. The class is declared (in our simple object-oriented language) as follows:

```
class Bag extends object {
  field a: array of int ;
  field n: int ;
  invariant self.a ≠ null ∧ 0 ≤ self.n ∧ self.n ≤ length(self.a) ;
  component self.a ;
  :
}
```

We write **self** for the implicit receiver parameter. Our arrays are like those in C# and Java: they are references to sequences of elements. The **component** declaration specifies a set of references that, in the steady state of a bag object, are owned by the bag (the reason for this particular declaration is explained below). The **invariant** declaration specifies a part of the internal consistency of the *Bag* representation.

Let's consider an instance method that inspects a bag without modifying it:

```
method Contains(x: int) returns (b: bool)
  requires self.inv = Bag
  modifies
  ensures true
{
  var i: int in
    b := false ; i := 0 ;
    while i < self.n do
      b := b ∨ (self.a[i] = x) ;
      i := i + 1
    end
  end
}
```

The method requires the bag's invariant to hold on entry, claims that nothing is modified, and allows an arbitrary return value (reflecting the fact that we are aiming only for a partial specification, like in ESC [4], not a full functional-correctness specification). According to our methodology, the condition $\text{self}.inv = Bag$ implies that the invariant declared in Bag (and also any invariant declared in the superclass `object`) holds. The body of the method can therefore rely on the condition $\text{self}.n \leq \text{length}(\text{self}.a)$, which together with the guard $i < \text{self}.n$ ensures that the array reference $\text{self}.a[i]$ does not overstep the array index bounds.

In our simple language, the constructors in C \sharp and Java are broken up into an allocation operation (`new`, which produces a new exposed object) and an explicit call to an initialization routine. The latter is just an *object procedure* (which in C \sharp would be called a *non-virtual method*). The class Bag provides the following initializing routine, which establishes Bag 's invariant:

```

proc FromArrayElements(e: array of int)
  requires  $\text{self}.exposed \wedge \text{self}.inv = \perp \wedge e \neq \text{null}$ 
  modifies  $\text{self}.\{Bag\}, \text{self}.inv$ 
  ensures  $\text{self}.inv = Bag$ 
{
   $\text{self}.ObjectInit()$  ;
   $\text{self}.n := \text{length}(e)$  ;
   $\text{self}.a := \text{new int}[\text{self}.n]$  ;
   $\text{self}.a.CopyFrom(e, 0, \text{self}.n)$  ;
  pack self as Bag
}

```

The precondition essentially says that `self` is a just-allocated object. The term $\text{self}.\{Bag\}$ in the modifies clause grants the license to modify all fields declared in Bag or in its superclasses (not counting the special fields *exposed* or *inv*). The modifies clause and postcondition also state that the object procedure returns after having established all of `self`'s invariants down to Bag .

The body of the initialization routine starts by calling an initialization routine of the superclass (here called *ObjectInit*). The `pack` statement at the end of the body has the effect of the following pseudo-code, here described more generally for `pack o as T`:

```

assert  $o \neq \text{null} \wedge o.exposed \wedge o.inv = \text{super}(T)$  ;
assert  $Inv(T, o)$  ;
foreach  $p \in Comp(T, o)$  do
  if  $p \neq \text{null}$  then
    assert  $p.exposed \wedge p.inv = \text{type}(p)$  ;
     $p.exposed := \text{false}$ 
  end
end
 $o.inv := T$ 

```

where the `assert` statements indicate conditions that are checked, $\text{super}(T)$ denotes the immediate superclass of T (or \perp if T is `object`), and $Inv(T, o)$ and $Comp(T, o)$ denote the invariants and components, respectively, declared in class T for the object o , and function *type* gives the dynamic type of an object.

Next, let's consider a method that updates a bag:

```

method Add(x: int)
  requires self.exposed  $\wedge$  self.inv = Bag
  modifies self.n, self.a
  ensures true
{
  unpack self from Bag ;
  if self.n = length(self.a) then
    var t: array of int in
      t := new int[2 · self.n + 1] ;
      t.CopyFrom(self.a, 0, self.n) ;
      self.a := t
    end
  end ;
  self.a[self.n] := x ;
  self.n := self.n + 1 ;
  pack self as Bag
}

```

Typical of an update method, this method body is bracketed by **unpack** and **pack**, where a statement **unpack** *o* **from** *T* has the effect of the following pseudo-code:

```

assert o  $\neq$  null  $\wedge$  o.exposed  $\wedge$  o.inv = T ;
o.inv := super(T) ;
foreach p  $\in$  Comp(T, o) do
  if p  $\neq$  null then p.exposed := true end
end

```

Note that method *Add* has the precondition *self.exposed*, which allows its implementation to perform **unpack** and **pack** operations on *self* and to modify the fields of *self*. The implementation also modifies the state of *self.a* (namely, its array elements). This modification requires *self.a.exposed*, but it would not be good information-hiding practice explicitly to give *self.a.exposed* in the precondition of *Add*. It is for this reason that *self.a* is listed in a **component** declaration, for then the **unpack** statement establishes *self.a.exposed*. Being a component also means that, in the steady state of the bag *self*, the array *self.a* is owned by *self*. In other words, the array underlying a bag is not used as a component by any other object.

Our methodology and specifications are powerful enough to indicate whether or not a parameter is *captured* by a method, that is, if the method takes over the ownership of the parameter. This is illustrated by the following initialization routine:

```

proc FromArray(e: array of int)
  requires self.exposed  $\wedge$  self.inv =  $\perp$   $\wedge$ 
    e  $\neq$  null  $\wedge$  e.exposed  $\wedge$  e.inv = type(e)
  modifies self.{Bag}, self.inv, e.exposed
  ensures self.inv = Bag
{
  self.ObjectInit() ;
  self.n := length(e) ;
  self.a := e ;
  pack self as Bag
}

```

The **pack** statement at the end of the routine body requires all components, which includes `self.a` to be exposed, so that they can be changed into being owned by the bag `self`. The precondition of the object procedure facilitates this, and the modifies clause gives the license to return from the routine having changed the value of `e.exposed`. The modifies clause tells callers that, although they can retain the reference `e`, they are no longer able to own `e`. Note also that our methodology detects the error of trying to implement routine `FromArrayElements` by the body of `FromArray` (because the precondition of **pack** would fail).

Of course, this example gives only the gist of the specification methodology. We do not yet have enough experience to say how powerful our approach is, but look forward to a fruitful discussion at the workshop.

3 Technical Approach

To define the semantics of our object-oriented language in the full paper, we translate it into a more primitive language (cf. [11]), which has a semantics defined by *weakest preconditions* [5, 15]. An object-oriented program is deemed correct if and only if the resulting primitive program is correct. In this section, we make a few remarks about the full system.

We require a kind of *prefix closure* property of declared components: for any subexpression `p.f` occurring in $Comp(T, \mathbf{self})$ or $Inv(T, \mathbf{self})$, either `p` is the literal `self` or `p` occurs in $Comp(T, \mathbf{self})$. Due to space limitations in this extended abstract, we omit here our treatment of array elements in **invariant** and **component**.

Our programming language guarantees the following two *system invariants* (stated informally in Section 1):

0. $(\forall o, T, S \bullet o.inv = T \wedge T <: S \Rightarrow Inv(S, o) \wedge (\forall c \bullet c \in Comp(S, o) \wedge c \neq \mathbf{null} \Rightarrow \neg c.exposed))$
1. $(\forall o \bullet \neg o.exposed \Rightarrow o.inv = type(o))$

where `o` ranges over non-null allocated objects, `T` and `S` range over object types (which does not include \perp), and `c` ranges over expressions. These system invariants always hold, because:

- any newly allocated object `o` satisfies $o.exposed \wedge o.inv = \perp$,
- the only statements that directly modify `inv` and `exposed` are **pack** and **unpack**, which maintain the system invariant,
- the value of $Inv(T, o)$ can be changed only by changing the value of some subexpression `p.f` in $Inv(T, o)$; note that if `p` is `o`, then `p.f` can be changed only if `o.exposed`; and note that if `p` is not `o`, then `p.f` is in $Comp(T, o)$, so `p.exposed` only if `o.inv` is a proper superclass of `T`.

The notion of objects being, at any moment, either exposed or owned gives us a way to interpret modifies clauses without the need for other abstraction features. We interpret a modifies clause `M` of any method as giving the license to modify the heap, subject to the following constraint, relating the pre-state and post-state of the method:

$$(\forall o, f \bullet o.f = \mathbf{old}(o.f) \vee o \in \mathbf{old}(M) \vee \neg \mathbf{old}(o.exposed))$$

where o ranges over non-null objects allocated in the pre-state and f ranges over field names. The first two disjuncts are standard, and say that either $o.f$ is not changed or $o.f$ is allowed to be modified according to the modifies clause (as interpreted in the pre-state). The third disjunct is new, and says that fields of owned objects may change during the method call.

4 Discussion

We are embarking on a project to write an invariant checker for the .NET platform and are considering the presented methodology as the foundation of the checker. Our full paper describes more details, but more needs to be done:

- *Expressive power of the language.* So far, we have only made a rudimentary investigation to see what kind of object-oriented programs we can verify. We expect our invariant declarations to be as easy to use as those in ESC/Java [7], but here one needs to explicitly give preconditions about invariants. Being explicit about that seems to help our system (unlike ESC/Java) to be sound. Constrained by soundness, our system is still flexible, in part due to the fact that one can express that only certain invariants hold (a feature not present in ESC/Java). Also, as we formulated it, packing succeeds only if the component graph (meaning the graph with objects as vertices and declared components as edges) from the object being packed is a tree. However, we think that useful relaxations of this rule exist.
- *Interdependence of effects and uniqueness.* Our system allows the reading of arbitrary objects, but allows updates only to exposed objects. One obtains this *exposed* bit from preconditions, newly allocated object, or, most frequently, by declaring objects to be components. This restriction is similar to proposals that allow updates on objects reachable from unique pointers, but there are interesting differences. Several other techniques also forbid reading or have to account in their analysis for reading, as done, for example, in Fugue [3] and in work by Boyland [1]. It would be interesting to further understand any relation between our allowing reading and the readonly references in universe types [14]. We expect that our technique can be extended to deal with richer heap structures as well.
- *Theory and practice.* We intend to implement the system, and may have a system running before the workshop starts. We also hope to do a formal proof of soundness.
- *Inference.* We believe that a practical system can only be realized if it supports some inference of specification fragments. Given class invariants, the existing approach seems well suited to infer some components, preconditions, and inductive invariants.

References

0. Amal Ahmed and David Walker. The logical approach to stack typing. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, volume 38, number 3 in *SIGPLAN Notices*, pages 74–85. ACM, March 2003.
1. John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice & Experience*, 31(1):533–553, January 2001.

2. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.
3. Robert DeLine and Manuel Fähndrich. The Fugue protocol checker: Is your software baroque? Submitted manuscript, 2003.
4. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
5. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.
6. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 13–24. ACM, May 2002.
7. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
8. Bart Jacobs. Objects and classes, co-algebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Oriented Programming with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publications, 1996.
9. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
10. K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 246–257. ACM, May 2002.
11. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.
12. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.
13. Peter Müller and Arnd Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.
14. Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, FernUniversität Hagen, 2001.
15. Greg Nelson. A generalization of Dijkstra’s calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.
16. David Walker, Karl Cray, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.