Reasoning about Object Structures Using Ownership

Peter Müller

ETH Zurich, Switzerland
Peter.Mueller@inf.ethz.ch

Abstract. *Many well-established concepts of object-oriented programming work for individual objects, but do not support object structures. The development of a verifying compiler requires enhancements of programming theory to cope with this deficiency.*
In this paper, we support this position by showing that classical specification and verification techniques support invariants for individual objects whose fields are primitive values, but are unsound for invariants involving more complex object structures.
We have developed an ownership model, which allows one to structure the object store and to restrict reference passing and the operations that can be performed on references. We use this model to generalize classical object invariants to cover such object structures. We summarize the state of our work and identify open research challenges.

1 Introduction

Programming theory encompasses, among other fields, language semantics, program logics, and specification techniques. It provides the foundation for understanding how programs work. Together with theorem proving technology, programming theory forms the basis of program verification, in particular, of program verifiers such as the verifying compiler [17].

Programming theory has been advanced to cope with new developments in programming languages and programming practice: Interface specifications in terms of pre- and postconditions, frame axioms, and invariants describe the behavior of methods and classes [27]. Abstraction functions [16] enable implementation-independent specifications of program behavior. Subtyping and dynamic method binding are addressed by behavioral subtyping [26,20]. Program logics cover a variety of programming language features [1,31]. However, despite these achievements, programming theory still falls behind programming practice.

This position paper describes one aspect of a larger effort to advance programming theory in order to improve tool-assisted verification of realistic programs. Our work focuses on modular specification and verification of object-oriented programs. *Modular* verification means that a class can be verified based on its implementation and the specifications of all classes it uses, but without knowing its subclasses and clients. Modularity is a prerequisite for the scalability

of verification techniques and tools, and for applying them to software libraries. By contrast, non-modular verification techniques require one to re-verify a class, say, a string class, in every context where it is (re-)used, which is not practical.

One of the major shortcomings of programming theory for object-oriented programming is summarized by the following position:

> *Many well-established concepts of object-oriented programming work for individual objects, but do not support object structures. The development of a verifying compiler requires enhancements of programming theory to cope with this deficiency.*

In this paper, we support this position by discussing one particularly important concept, namely object invariants. We illustrate the problems and present ownership as a general solution to this class of problems.

For simplicity, we consider a restricted programming language here. We use a language similar to C# and Java, but omit multi-threading, inheritance, and static class members. However, the presented techniques do not rely on these restrictions [18,23,22,28].

Overview. In Section 2 we describe the classical technique for reasoning about invariants and its limitations. Section 3 presents a modular verification technique for invariants over object structures based on an ownership model. We summarize our progress so far and identify open research challenges in Section 4.

2 Classical Invariants and their Limitations

Invariants are predicates that specify what states of an object are consistent [16]. For example, the invariant of the `List` class, near the top of Fig. 1, states several such properties, including that the `array` field is always non-null and that the array holds non-negative numbers. Thus, when calling `add`, for example, the expression `array.length` cannot cause a null pointer exception.

The invariant semantics used by classical reasoning techniques [25,26,27] is that each object has to satisfy its invariant in the pre- and poststate of each exported method. To enforce this property, classical techniques require one to prove that each exported method preserves the invariant of its receiver object. For this proof, one may assume that in the prestate of the method execution the precondition of the method holds and that all allocated objects satisfy their invariants. For constructors, one has to show that the invariant of the new object is established.

The classical techniques assume that a method can break only the invariant of its receiver object. Therefore, they are sound for invariants of individual objects whose fields are primitive values, such as points with integer coordinates. However, since they do not impose proof obligations on the invariants of other objects, these techniques do not support invariants of more complex object structures.

Abstraction Layering is not Sound. Classical techniques do not support invariants of layers implemented on top of `List`. For example, the invariant of class `BagWithMax` in Fig. 2, which says that no element of the list is larger than a given upper bound, is generally not preserved by `List`'s `add` method. That is, the classical technique's assumption that a method can break only the invariant of its receiver object is not valid for invariants that depend on the state of several objects. In particular, the call to `add` in `BagWithMax`'s `insert` method temporarily violates the invariant of the `BagWithMax` object if `k` is greater than `maxElem`. The invariant is restored by the last statement of `insert`. Even if one would require `add` to preserve `BagWithMax`'s invariant, this example would not be handled modularly, since modular verification of a class implies not considering its clients during its verification.

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

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

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

  //@ assignable array, n;
  public void resize()              { /* ... */ }
  public List()                     { array = new /*@ rep @*/ int[10]; }
  public void addElems(int[] elems) { /* ... */ }
  // other methods omitted.
}
```

Fig. 1. Implementation and JML specification of an array-based list. Annotation comments start with an at-sign (@), and at-signs at the beginning of lines are ignored. The array object is part of the encapsulated internal representation of the list, indicated by the `rep` annotation. The `spec_public` annotation allows fields with any access modifier to be mentioned in public specifications.

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

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

  //@ requires k>=0;
  public void insert(int k) {
    theList.add(k);  // temporary invariant violation if k > maxElem
    if (k > maxElem) { maxElem = k; }
  }
  // other methods and constructors omitted.
}
```

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

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

Another way to view this soundness problem is that the classical invariant semantics is too strong for invariants over layered object structures. The class `List` cannot modularly know enough to establish the invariant of a class, `BagWithMax`, that it does not know about. Note that this problem is not due to aliasing. It occurs even if `BagWithMax` objects have unique references to their `List` objects.

Mutable Subobjects are not Sound. For example, the invariant in class `List` of Fig. 1 is not supported by classical techniques, because it refers to locations in the underlying array object. If a reference to the array could be exposed to other objects, then any method of the program could use such a reference to break `List`'s invariant [10,25]. Such an alias could occur by rep exposure or by capturing as illustrated by the version of `addElems` in Fig. 3. With that version, the code fragment in Fig. 4 would violate `List`'s invariant. This could happen even if the classical proof technique was used to prove the correctness of all of

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

Fig. 3. A questionable implementation of the `List` method `addElems` that stores the argument array into the `array` field.

`List`'s methods. This example shows that a sound technique must either restrict invariants that depend on subobjects in lower abstraction layers, or it must control aliasing and, in particular, modifications of such subobjects.

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

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

3 Ownership-Based Invariants

Ownership allows one to structure the object store and to restrict reference passing and the operations that can be performed on references. We use ownership also to control the dependencies of invariants and to define a weaker semantics for invariants that allows layering.

3.1 Ownership Model

Ownership organizes objects into *contexts*: Each object is owned by at most one other object, called its *owner*. A context is the set of all objects with the same owner. The set of objects without owner is called the *root context*. The contexts of a program execution form a tree, where the context of all objects with owner

X is a child of the context containing X. The context tree is rooted in the root context.

Our ownership model enforces the *owner-as-modifier* property: All modifications of an object, X, must be initiated by X's owner. That is, X can be referenced by any other object, but reference chains that do not pass through X's owner must not be used to modify X [22,28]. Therefore, owners can control modifications of owned objects.

The ownership relation is expressed in programs by the ownership modifier `rep`, which can be used in field declarations and object creation expressions. In class `List` (Fig. 1) the `rep` keyword indicates that the array referenced by `array` and the array created in the constructor are owned by `this`.

Ownership and the owner-as-modifier property can be enforced by type systems or by standard verification techniques [12]. For instance, our Universe type system [28] would forbid the assignment to `array` in Fig. 3 and require copying `elems` to avoid the unwanted alias.

3.2 The Ownership Technique

The ownership model allows one to generalize the classical technique to invariants over layered object structures. To avoid the soundness problems described in Sec. 2, we use the hierarchical structure of the ownership model to refine the classical invariant semantics.

Admissible Invariants. The ownership technique allows the invariant of an object, X, to depend on fields of X (like the classical technique) and on fields of objects owned by X. The invariant of class `List` is an ownership-based invariant because it depends on the fields of `this` (`array` and `n`) and on fields of the array owned by `this` (`array.length` and `array[i]`).

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

Semantics of Invariants. In Section 2, we showed that invariants over layered object structures may not hold in the pre- and poststates of all exported methods. For example, a `BagWithMax` object needs to temporarily violate its invariant when changing its underlying `theList` object. To allow such violations, we weaken the invariant semantics from the classical technique.

The weakened semantics allows a method executed on a receiver object, X, to violate the invariants of all objects in ancestor contexts of the context containing X. In particular, the method is allowed to violate the invariants of X's transitive owner objects. For instance, method `add` executed on a `List` object is allowed to violate the invariant of the owning `BagWithMax` object.

Ownership Proof Technique. Like the classical proof technique, the ownership proof technique requires one to prove that each exported method preserves the invariant of its receiver object. For this proof, one may assume that in the prestate of the method execution (1) the precondition of the method holds and (2) those allocated objects that are in the context of the receiver object or its descendants satisfy their invariants. However, a method must not assume that the (transitive) owners of the receiver object satisfy their invariants, which corresponds to the refined invariant semantics.

To illustrate how to use the ownership proof technique, consider `BagWithMax`'s `insert` method. For the call to `List`'s `add` method we may assume that the method preserves the invariant of `theList`. However, since the `BagWithMax` object `this` is the owner of the receiver of this call to `add`, its invariant might be broken by the call. To show that the `insert` method preserves this invariant, we use the postcondition of `add` to derive that the list after the call contains exactly the elements before the call plus the new element `k`. If `k` happens to be a new maximum in the list, then `BagWithMax`'s invariant is violated after the call, but reestablished by the subsequent assignment to `maxElem`. Therefore, the invariant of `this` is preserved.

As can be seen from the example above, responsibility for verifying invariants is divided. A method's implementor is responsible for the objects (transitively) owned by the method's receiver object, but its calling method is responsible for other objects.

This ownership proof technique is modular and sound [22,28,30].

4 Progress so far and open Research Challenges

We have developed the ownership technique for object invariants in cooperation with Gary Leavens, Rustan Leino, and Arnd Poetzsch-Heffter [22,28,30]. To handle implementations that are not supported by the ownership model such as mutually recursive data structures, we combined ownership-based invariants with so-called visibility-based invariants that gain modularity by imposing certain visibility requirements on fields [4,22,24,30]. We have also extended our methodology to static class invariants [23] and adapted it to the verification of frame properties [29]. The combination of these techniques can handle many interesting implementations.

We have developed two approaches to specifying the ownership relation and enforcing the owner-as-modifier property. The Universe type system [12,28] expresses ownership by extended type information and checks the owner-as-modifier property as part of type checking. An alternative approach [22] encodes ownership by a specification-only field that can be used in interface specifications. The owner-as-modifier property is enforced by a programming methodology that restricts how objects can be modified.

Although we have made significant progress with ownership-based verification, there are a number of open research challenges. We summarize these challenges in the following.

Ownership. Our ownership model can handle realistic applications. However, there are common implementation patterns that cannot directly be expressed, for instance, several objects sharing and modifying a common representation. We plan to generalize our ownership model to allow more implementations, in particular, multiple ownership and ownership transfer. We will also study how implementation patterns that are not directly supported can be rewritten to follow the ownership model.

To reduce the overhead of writing ownership annotations, we are working on ownership inference. Besides classical type inference techniques, we study runtime inference [13] to infer ownership relations. The results so far are promising. A major application of ownership inference will be to run case studies to investigate how common ownership relations are.

Specification Features. An important topic for future work is invariants over model (specification-only) fields [19,21], which are useful to describe properties of data structures without referring to their concrete implementation.

History constraints [26] suffer from the same problems as the classical invariant technique when applied to object structures. We plan to extend ownership-based techniques to history constraints and to more general temporal constraints.

Specification languages like Eiffel and JML allow method calls in interface specifications. Methods that can be called in specifications must be pure, that is, side-effect free. Therefore, they can be formally modeled by mathematical functions. Especially for recursive functions, ownership can help to show that the functions are well-defined [8]. We plan to support method calls in specifications in the Boogie tool [3], which is being developed at Microsoft Research.

Automation. Practical applications of program verification require a high degree of automation. Research on automated verification has mainly focused on automated verification tools [11,15] and automated theorem provers [9].

We investigate combinations of classical logic-based reasoning with extended type systems [14,28], abstract interpretation [7], and static analyses. For instance, we plan to develop static, modular analyses for purity and frame properties of methods. These analyses build on the ownership structure to simplify the pointer and escape analysis used by related approaches [32].

5 Conclusions

We have illustrated the problems of reasoning about object structures by discussing object invariants. We have summarized the ideas of ownership-based verification, which allows one to reason about layered object structures in a modular way. Details of this approach are presented in earlier papers [22,30].

Ownership allows one to structure the object store systematically. Structuring is important for many important applications. For instance, ownership has also been applied successfully to the modular verification of frame properties [29], static class invariants [23], reasoning about multi-threaded programs

[5,18], confinement of internal representations of data structures [6], and proving representation independence [2]. Therefore, we believe that ownership is a fundamental principle of programming theory.

Parts of our work on the foundations of ownership and ownership-based verification have been implemented in JML [19], in the semi-automated Java Interactive Verification Environment (JIVE), and in the fully automated verification tool Boogie [3]. One important aspect of future work is to use these tools for non-trivial case studies.

References

1. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT)*, volume 1214 of *LNCS*, pages 682–696. Springer-Verlag, 1997.
2. A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages*, pages 166–177. ACM, 2002.
3. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented progra ms. Submitted. Available from `http://research.microsoft.com/~leino/papers/krml160.pdf`, 2006.
4. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen and C. Shankland, editors, *Mathematics of Program Constructions*, volume 3125 of *LNCS*. Springer-Verlag, 2004.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.
6. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of programming languages (POPL)*, pages 238–252. ACM Press, 1977.
8. A. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, June 2006. To appear.
9. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
10. D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research Report 156, Digital Systems Research Center, 1998.
11. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Digital Systems Research Center, 1998.
12. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.
13. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.
14. M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Object-oriented programing, systems, languages, and applications (OOPSLA)*, pages 302–312. ACM Press, 2003.

15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, 2002.
16. C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
17. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.
18. B. Jacobs, K. R. M. Leino, and W. Schulte. Verification of multithreaded object-oriented programs with invariants. In *Specification and Verification of Component-Based Systems (SAVCBS)*, pages 2–9, 2004. Technical report 04-09, Department of Computer Science, Iowa State University.
19. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
20. G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *SIGPLAN*, pages 212–223. ACM, 1990.
21. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.
22. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
23. K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *LNCS*, pages 26–42. Springer-Verlag, 2005.
24. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, 2002.
25. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
26. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.
27. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
28. P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.
29. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.
30. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 2006. Accepted for publication. Also available as TR 424 of the Department of Computer Science, ETH Zurich.
31. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.
32. A. Salcianu and M. Rinard. Purity and side effect analysis for java programs. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.